Datalog on Infinite Structures

DISSERTATION

zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen
Fakultät II
Humboldt-Universität zu Berlin

von
Herr Dipl.-Math. Goetz Schwandtner
geboren am 10.09.1976 in Mainz

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Christoph Markschies

Dekan der Mathematisch-Naturwissenschaftlichen
Fakultät II:
Prof. Dr. Wolfgang Coy

Gutachter:

1. Prof. Dr. Martin Grohe
2. Prof. Dr. Nicole Schweikardt
3. Prof. Dr. Thomas Schwentick

Tag der mündlichen Prüfung: 24. Oktober 2008

Abstract

Datalog is the relational variant of logic programming and has become a standard query language in database theory. The (program) complexity of datalog in its main context so far, on finite databases, is well known to be in EXPTIME. We research the complexity of datalog on infinite databases, motivated by possible applications of datalog to infinite structures (e.g. linear orders) in temporal and spatial reasoning on one hand and the upcoming interest in infinite structures in problems related to datalog, like constraint satisfaction problems:

Unlike datalog on finite databases, on infinite structures the computations may take infinitely long, leading to the undecidability of datalog on some infinite structures. But even in the decidable cases datalog on infinite structures may have arbitrarily high complexity, and because of this result, we research some structures with the lowest complexity of datalog on infinite structures: Datalog on linear orders (also dense or discrete, with and without constants, even colored) and tree orders has EXPTIME-complete complexity.

To achieve the upper bound on these structures, we introduce a tool set specialized for datalog on orders: Order types, distance types and type disjoint programs. The type concept yields a finite representation of the infinite program results, which could also be of interest for practical applications. We create special type disjoint versions of the programs allowing to solve datalog without the recursion inherent in each datalog program.

A transfer of our methods shows that constraint satisfaction problems on infinite structures occur with arbitrarily high time complexity, like datalog.

Keywords:
Datalog, Infinite Structures, Complexity, Decidability

Zusammenfassung

Datalog ist die relationale Variante der logischen Programmierung und ist eine Standard-Abfragesprache in der Datenbankentheorie geworden. Die Programmkomplexität von Datalog im bisherigen Hauptanwendungsgebiet, auf endlichen Strukturen, ist bekanntermassen in EXPTIME. Wir untersuchen die Komplexität von Datalog auf unendlichen Strukturen, motiviert durch mögliche Anwendungen von Datalog auf unendlichen Strukturen (z.B. linearen Ordnungen) im zeitlichen und räumlichen Schliessen, aber auch durch das aufkommende Interesse an unendlichen Strukturen bei verwandten theoretischen Problemen, wie Constraint Satisfaction Problems (CSP):

Im Gegensatz zu endlichen Strukturen können Datalog-Berechnungen auf unendlichen Strukturen unendlich lange dauern, was zur Unentscheidbarkeit von Datalog auf unendlichen Strukturen führen kann. Aber auch in den entscheidbaren Fällen kann die Komplexität von Datalog auf unendlichen Strukturen beliebig hoch sein. Im Hinblick auf dieses Ergebnis widmen wir uns dann unendlichen Strukturen mit der niedrigsten Komplexität von Datalog: Wir zeigen, dass Datalog auf linearen Ordnungen (auch dichte und diskrete, mit oder ohne Konstanten und sogar gefärbte) und Baumordnungen EXPTIME-vollständig ist.

Für die Bestimmung der oberen Schranke werden Werkzeuge für Datalog auf Ordnungen eingeführt: Ordnungstypen, Abstandstypen und typdisjunkte Programme. Die Typkonzepte liefern eine endliche Beschreibung der unendlichen Programmergebnisse und könnten auch für praktische Anwendungen von Interesse sein. Wir erzeugen spezielle typdisjunkte Programme, die sich ohne Rekursion lösen lassen.

Ein Transfer unserer Methoden auf CSPs zeigt, dass CSPs auf unendlichen Strukturen mit beliebig hoher Zeitkomplexität vorkommen, wie Datalog.

Schlagwörter:
Datalog, Unendliche Strukturen, Komplexität, Entscheidbarkeit

Acknowledgements

I want to thank my thesis advisor Martin Grohe for guiding me through the world of datalog on infinite structures in many valuable discussions and with many helpful suggestions and comments on my ideas. I thank Nicole Schweikardt for several interesting discussions leading to new ideas and views on the research presented in this thesis.

I am grateful to Mark Weyer for his careful proofreading of most of my ideas and pointing out mistakes in the proofs, and of course for his suggestions about possible solutions. Thanks go to Thomas Gottron and Kord Eickmeyer for proofreading parts of this thesis.

Although he has not participated in the work at this thesis directly, I want to thank my former advisor Clemens Lautemann. With his interesting lectures and seminars and in numerous discussions he has aroused my curiosity about questions in computer science, especially in complexity theory and logics. Sad, that he passed away so soon!

I am thankful to my colleagues at the computer science department at the Johannes Gutenberg-University in Mainz, especially Elmar Schömer, to enable me to finish my research project after the sudden loss of my former advisor and also Thomas Schwentick for suggesting Martin as new advisor.

Last but not least I thank my parents for their continuing support.

Contents

Chapter 1

Introduction

Database theory is an important part of theoretical computer science and investigates the foundations of databases and database systems. One main aspect of database theory is the complexity of queries whose research is not only interesting for practical applications, but also leads to results in other fields of theoretical computer science. A query language which has been studied extensively in database theory and which has become a standard tool is datalog (see, e.g., [Var82; Ull88a; AHV95]). Datalog is a query language based on logic programming and inherits the simple program structure from this paradigm: Each datalog program consists of a finite set of rules consisting of relational atoms. Some of the relations, the intensional databases (IDBs), are part of the constant input structure, while the other relations, the extensional databases (EDBs), are computed by applying the program rules. The computation of datalog is carried out in stages, starting with empty IDB relations and for each stage applying a rule to the previous stage, until a fixed point is reached. For example, let $G = (V, E)$ be a directed graph with vertex set V and edge relation E, then the following two rule program will compute the transitive closure T of the edge relation:

$$\begin{aligned} T(x, y) &\leftarrow E(x, y). \\ T(x, y) &\leftarrow T(x, z),\ E(z, y). \end{aligned}$$

While the first rule copies the edges into the transitive closure, the second rule is a recursive rule, in each application adding pairs of vertices to T that are connected by a path one edge longer than for all existing pairs of vertices in T.

This simple example demonstrates that datalog is capable of recursion, which other query languages, like first order queries or the relational algebra do not offer. On finite structures, a datalog computation will always terminate, since there may only be finitely many rule applications adding new tuples to the IDB relations, hence the fixed point is reached after finitely many stages. But it is less obvious how long such a computation may take, which motivates a complexity analysis as in [DEGV97]. For database queries different complexity measures are common, depending on the choice of input. For *data complexity*, the query (or program) is considered constant, while the database is the input and the running time determined depending on the database

size. *Program complexity* (or expression complexity) measures the complexity for constant databases depending on the query only, and for *combined complexity*, both the database and the query are part of the input. Dantsin et. al. [DEGV97] showed that on finite databases, datalog has PTIME-complete data complexity, while the program complexity and combined complexity are EXPTIME-complete.

In this thesis, we investigate the complexity of datalog on infinite structures. At a first glance, only finite structures are of interest for database applications implemented on computer systems, as practical problems involve finite amounts of data only. But temporal and spatial reasoning naturally include infinite structures for modeling data, as time in temporal reasoning is usually modelled as an infinite linear order, sometimes dense, sometimes discrete. Finite representations of temporal or spatial data may lead to a noticeable loss in accuracy, which is why databases and database query languages working directly on infinite structures are of interest. Our focus on datalog on infinite linear orders does not only apply to the background structures for temporal reasoning, but easily transfers to the setting of intervals: Allen's interval algebra [All82] is a standard tool in temporal reasoning using operations on (time) intervals and we show that our results for datalog on linear orders apply to this setting in a natural way.

Of course, applications from temporal or spatial reasoning are not the only motivation to consider datalog on infinite structures. On infinite structures, datalog has much more computational power than on finite structures, making it worthwhile to study the complexity of datalog on infinite structures. As in the applications mentioned above, we will regard the infinite structure as a fixed "background" structure, thus restrict ourselves to program complexity. For a meaningful concept of data complexity or combined complexity, a finite representation of infinite structures is needed. Most of our results are independent of the representation, leading to a more general approach. Our setting of program complexity on a fixed background structure is not as restrictive as it may seem at first: Using a suitable background structure (as a linear order), a finite database (consisting of, e.g., order atoms) may be included into the program, leading to some weak form of combined complexity.

To formalize the term "complexity of datalog", we consider the problem DATALOG-NONEMPTINESS($\mathcal{A}$) for fixed infinite structures $\mathcal{A}$:

Given: Datalog program Π on $\mathcal{A}$ and an IDB P of Π.

Question: Is the relation P computed by Π empty?

As seen above, on infinite structures the datalog computations may have infinitely many stages, reaching the fixed point in a countable infinite number of stages. Consider the above program for the transitive closure on an infinite directed graph $\mathcal{G}$: If there are simple paths of arbitrarily high lengths in $\mathcal{G}$, the computation will continue forever. If infinite computations may occur, it is no longer clear if datalog is decidable, even for DATALOG-NONEMPTINESS. One result of this thesis shows that undecidability on infinite structures is, in fact, the case:

DATALOG-NONEMPTINESS($\mathcal{A}$) is undecidable on infinite successor structures.

This result is a consequence of our proof showing that on successor structures datalog is universal in the sense of computation.

Therefore, for complexity investigations and even more for practical applications we have to be careful with the choice of structures. In the first part we generalize a result from [DEGV97] which shows the EXPTIME-completeness of datalog with constants on finite structures to show that on all structures (finite and infinite, with or without constants) on which two disjoint sets may be defined, DATALOG-NONEMPTINESS is EXPTIME-hard.

With undecidability at one end and a quite low lower bound at the other end, the question remains, what happens in between. We show that for each class C of the exponential hierarchy[1] there is an infinite structure $\mathcal{C}$ such that DATALOG-NONEMPTINESS($\mathcal{C}$) is contained in this class and hard for the class one level lower.

After this result, it seems interesting (especially with practical applications in mind) to identify some classes of structures with the lowest complexity of datalog on infinite structures, i.e. EXPTIME-complete. As it turns out, linear orders are an ideal candidate to investigate solving techniques for datalog using special properties of infinite structures, leading to this complexity. The main results of the second part can be summarized as:

DATALOG-NONEMPTINESS($\mathcal{A}$) is EXPTIME-complete on

1. dense linear orders $\mathcal{A} = (A, <)$ and dense linear orders with constants $\mathcal{A} = (A, <, c_1, \ldots, c_k)$
2. (arbitrary) linear orders $\mathcal{A} = (A, <)$ and $\mathcal{A} = (A, <, c_1, \ldots, c_k)$
3. colored linear orders $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ (with monadic relations $M_1, \ldots, M_m$)
4. tree orders $\mathcal{A} = (A, <)$ and $\mathcal{A} = (A, <, c_1, \ldots, c_k)$

The first result may not be surprising as dense linear orders allow quantifier elimination (see, e.g., [Hod97]), bringing formulae to a simpler equivalent shape. In terms of datalog quantifier elimination is mainly the elimination of variables occurring in the rule body only, and not the head IDB atom, which leads to some elimination of recursion. From this point of view, the second result (which actually includes the first as a special case) is, in fact, something unexpected. The class of linear orders includes discrete linear orders, which can be defined as transitive closure of successor structures, even by a short datalog program as seen above. This connection may lead to the impression that such similar structures as successor structures and discrete

[1] The complexity classes DTIME(2^{n^k}), DTIME($2^{2^{n^k}}$), DTIME($2^{2^{2^{n^k}}}$), ... of iterative exponential time complexity; see e.g. [Pap94].

orders behave similar and lead to a similar complexity of DATALOG-NONEMPTINESS. The results of this thesis mentioned above clearly show that this is not the case.

The result about colored orders is perhaps even more interesting for applications from temporal reasoning: The linear order serving as time-line as usual, the monadic relations can be used as flags for the state of a system under surveillance. For each instant t, the membership and non-membership of t in the monadic relations gives a description of the system and datalog enables us to ask queries setting different states in context.

The last result about tree orders mentioned above is a first glance from total orders to partial orders, showing how the methods relying on the (linear) order axioms can be applied to a setting where not all these axioms are satisfied.

With the above results we have a full classification of the complexity of DATALOG-NONEMPTINESS on linear orders by the arity of additional relations allowed. We have no relations (or nullary relations) in the case of linear orders and monadic relations in the case of colored orders. If we allow arbitrary binary relations (or even higher arity), we may add a successor relation, leading to undecidability for this case.

The toolbox introduced for deriving these upper bounds offers some methods which are also interesting by themselves:

1. An order type concept for datalog. We define order types as finite sets of atomic formulae over the vocabulary $\{<\}$ and show that finite sets of order types are sufficient to describe all IDBs created by a datalog program over dense linear orders.

2. For arbitrary orders, we extend the order types to distance types, built on atomic formulae of the form $(x <_d y)$ stating that $x < y$ has to be satisfied and that there have to be d different elements between x and y. We show that a finite set of distance types is sufficient to represent each stage of the IDBs in a computation of a datalog program.

3. To solve DATALOG-NONEMPTINESS($\mathcal{A}$) *without using recursion*, the datalog program is converted into a type disjoint form which incorporates all properties of the underlying structure $\mathcal{A}$ needed for datalog computations into the program.

The type concepts allow a finite representation of computation results, which could also be used to perform datalog computations over infinite structures using a computer. These concepts have some interesting advantages over existing approaches to datalog on infinite structures. Usually, datalog programs have to satisfy some safety conditions (see, e.g., [SV89; CGZ07]): The results of datalog queries always have to be finite, which can be guaranteed if each rule application only produces a finite result (weak safety) and all computations of a program terminate. Weak safety

also restricts the syntax of datalog programs: Each variable in the head IDB atom also has to appear in the body. Since we have a finite description via types, we do not need this restriction and other restrictions leading to a weakly safe program.

Termination, on the other hand, is closely linked to the boundedness problem. For a class $\mathcal{C}$ of structures, a program Π is bounded if there exists a constant c such that on each structure $A \in \mathcal{C}$ the computation of Π terminates after at most c steps. For the class of finite structures, the boundedness problem has received considerable attention (see, e.g., [GMSV93; CGKV88; Mar99]) and has proven to be undecidable even for a very restricted class of programs, like programs with a single recursive rule (see, e.g., [Abi89; HKMV95]). Using the distance type concept, we show the following interesting result:

On the class $\mathcal{C}$ of linear orders $\mathcal{A} = (A, <)$, datalog is uniformly bounded.

In this case, the bound is a computable function of the program length, hence a *uniform* bound. With this result it is also possible to compute the finite set of distance types describing the full computation of a datalog program.

In the last part of this thesis, we have a look at constraint satisfaction problems (CSPs), which are intimately linked with datalog (see, e.g., [FV99]). CSPs are widely used in computer science, as well in the center of theoretical research, as well as in applied fields, like artificial intelligence or bio-informatics. Starting with [BN03] infinite CSPs were introduced into current research and have been studied since.

We transfer some methods from datalog to CSP and maybe the most interesting result in this context is the following:

For each class C of the exponential hierarchy there is an infinite structure $\mathcal{C}$ such that CSP($\mathcal{C}$) is contained in this class and hard for the class one level lower.

1.1 Published Results

Parts of this thesis (mainly from Chapter 5) have been submitted for publication in [GS].

1.2 Outline of This Thesis

In **Chapter 2** we will define the tools and problems we need for our discussion, including some remarks about the representation issues involved in the work with infinite structures. As a look at applications in temporal reasoning, we will establish a link between datalog on linear orders and Allen's interval algebra.

The main tool introduced in **Chapter 3** is a generalized version of the universal Turing machine simulation by datalog programs from [DEGV97]. This machine simulation is used to show some lower bounds for the DATALOG-NONEMPTINESS and DATALOG-TUPLE, followed by **Chapter 4**, in which we use the Turing machine simulation from the preceeding chapter to derive some undecidability results. The

chapter is concluded by a construction of a hierarchy of structures for the complexity of DATALOG-NONEMPTINESS.

Chapter 5 introduces the type concept describing datalog computations on linear orders and EXPTIME upper bounds for the DATALOG-NONEMPTINESS on linear orders is derived using this type concept. For the example of dense linear orders, an algorithm is given, which demonstrates how to implement types in computations. In **Chapter 6**, a type concept for colored orders is introduced, which allows the transfer of some of the results from the preceeding chapter to show an EXPTIME upper bound on colored orders, i.e. orders with additional monadic relations. Chapters 5 and 6 are written independent of Chapters 3 and 4 and can be read without the knowledge of these chapters.

The methods considered so far are transferred to an application on tree orders, which are partial orders based on infinite trees in **Chapter 7**.

In **Chapter 8**, results from the preceeding chapters about orders are transferred to constraint satisfaction problems. Some (partially very large) upper bounds are shown with the methods of Chapters 5 and 6, and modifying results and ideas from Chapter 4, a hierarchy result for CSPs similar to the one in Chapter 4 is derived.

We will give a brief overview of each chapter at the beginning of the chapter and a conclusion at the end, followed by some open problems, which may lead to further research.

The results and interesting open questions are summarized in **Chapter 9**, which gives an overview of the most important results and open questions in the view of this thesis' author.

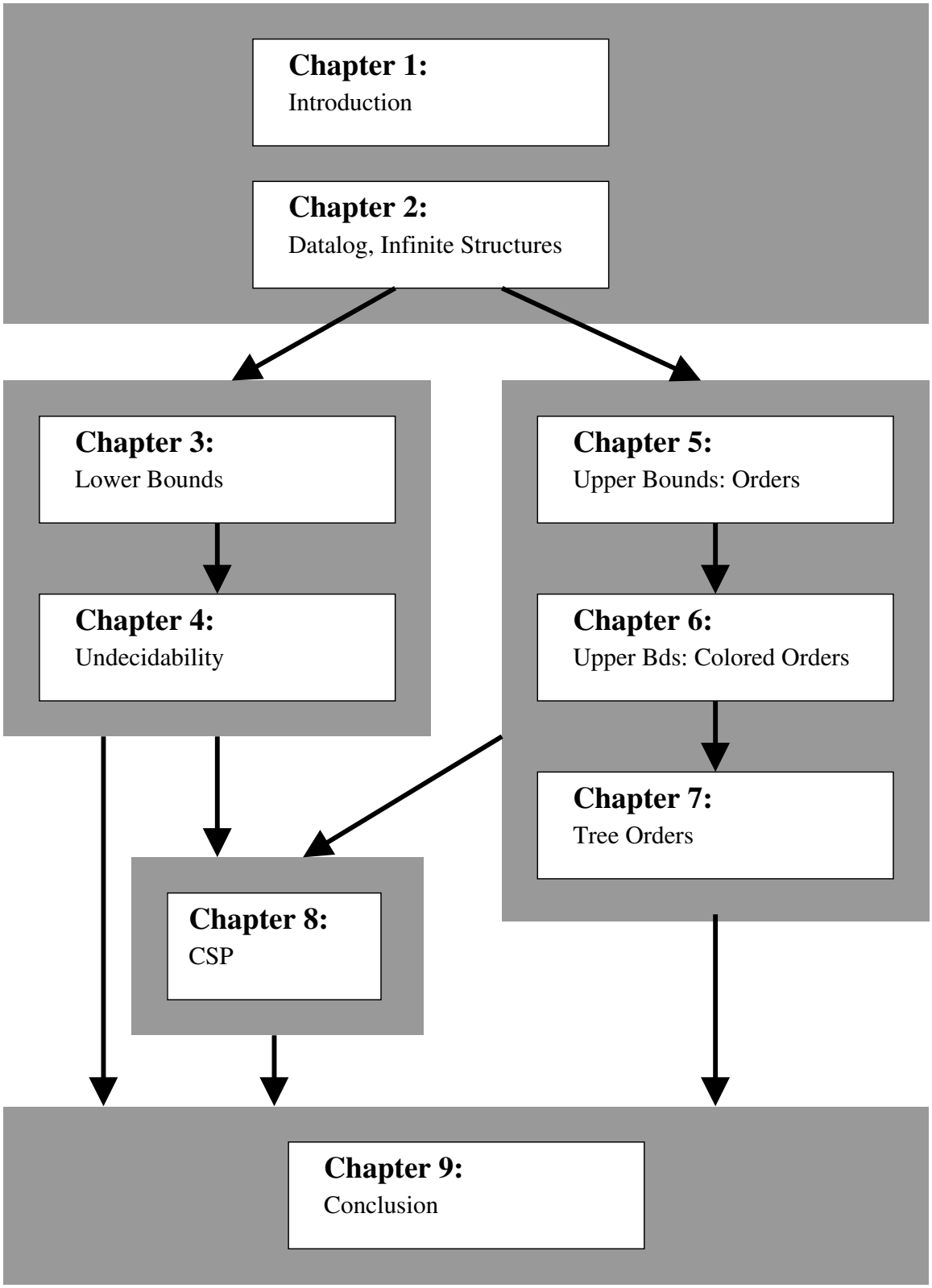


Figure 1.1: Dependencies among the chapters of this thesis.

Chapter 2

Datalog, Infinite Structures

Outline of this Chapter

In this chapter we will give a formal definition of datalog and the datalog problems in the scope of this thesis. We then define the infinite structures of interest for this thesis, namely orders and successor structures, and sketch different methods for representing infinite structures. We conclude this chapter with a closer look at datalog on Allen's interval algebra, an example application from the context of temporal and spatial reasoning.

2.1 Notations from Logics and Model Theory

We assume that the reader is familiar with the basic concepts of complexity theory, like the deterministic complexity classes PTIME (or P) and EXPTIME, logarithmic space reductions and completeness. For the presentation of our undecidability results we assume basic knowledge about the universal Turing machine model, the halting problem and reductions. An introduction of these fields can be found, e.g., in [GJ79] or [Pap94].

In the following sections we give a short definition of the logical notation used in this work. Although not used explicitly, a general knowledge of first order logic as found in introductional books about mathematical logic or model theory ([EFT98; Bur98; EF99; Hod97]) is useful for following this thesis, as well as some knowledge about inductive definitions and fixed points as introduced in [Mos74].

2.1.1 Structures

A *vocabulary* σ is a finite set of symbols of which some are *constant symbols* (usually denoted by $c, c_1, c_2, \dots$) and the others are *relation symbols* (usually denoted by $P, R, R_1, R_2, \dots$)[1]. A *relational vocabulary* is a vocabulary containing no constant

[1]Since we will deal with a purely relational context throughout this thesis, we omit function symbols here.

symbols. An *arity map* $\text{ar} : \sigma \to \mathbb{N}$ defines an arity for each relation symbol. A *structure* $\mathcal{A}$ *of vocabulary* σ (or shorter a σ*-structure*) consists of a *universe* A and for each constant symbol an interpretation $c^{\mathcal{A}} \in A$ and for each relation symbol R a relation $R^{\mathcal{A}} \subseteq A^{\text{ar}(R)}$ of arity $\text{ar}(R)$ as interpretation.

2.1.2 Homomorphisms

A *homomorphism* h from some σ-structure $\mathcal{A}$ with universe A to some σ-structure $\mathcal{B}$ with universe B is a map $h : A \to B$ preserving constants if present, $h(c^{\mathcal{A}}) = c^{\mathcal{B}}$ for all constant symbols $c \in \sigma$, and preserving relations: For each relation symbol $R \in \sigma$ and each tuple $\bar{a} = (a_1, \ldots, a_{\text{ar}(R)}) \in R^{\mathcal{A}}$, it holds that $h(\bar{a}) = (h(a_1), \ldots, h(a_{\text{ar}(R)})) \in R^{\mathcal{B}}$.

2.2 Datalog

Datalog is the relational sublanguage of logic programming. Datalog programs consist of function-free and negation-free horn clauses, which we will augment by constants in some cases, but only if mentioned explicitly.

An *atom* is an expression of the form $P(x_1, \ldots, x_k)$, where P is a k-ary relation symbol and $x_1, \ldots, x_k$ are variables. We admit 0-ary relation symbols.[2] In the following, we abbreviate tuples $(x_1, \ldots, x_k)$ by $\bar{x}$. A *datalog rule* is an expression ρ of the form

$$P\bar{x} \leftarrow Q_1\bar{y}_1, \ldots, Q_m\bar{y}_m,$$

where $P\bar{x}$, $Q_1\bar{y}_1, \ldots, Q_m\bar{y}_m$ are atoms. The tuples of variables $\bar{x}, \bar{y}_1, \ldots, \bar{y}_m$ need not be disjoint, and variables may occur several times in each tuple. Furthermore, the variables in $\bar{x}$ are not required to be among those in $\bar{y}_1, \ldots, \bar{y}_m$, which is a main difference to other definitions found in literature.[3]

The atom $P\bar{x}$ is the *head* of the rule, $Q_1\bar{y}_1, \ldots, Q_{m_\rho}\bar{y}_{m_\rho}$ is the *body*. A *datalog program* is a finite set of datalog rules. Relation symbols occurring in the head of a rule of a datalog program Π are called *intensional relation symbols* or *IDBs*; all other relation symbols are called *extensional relation symbols* or *EDBs*.

We say that a datalog program Π is *over* a structure $\mathcal{A}$ if the vocabulary of $\mathcal{A}$ contains all EDBs of Π and none of the IDBs. Π is a datalog program *over* a class C of structures if Π is a program over all $\mathcal{A} \in C$.

2.2.1 Datalog Computation

Let Π be a Datalog program over a structure $\mathcal{A}$. The *computation of* Π *over* $\mathcal{A}$ is carried out in stages in which the interpretation of the IDBs is computed; the

[2] A 0-ary relation either is empty, or it consists of the empty tuple ().

[3] The reason for this restriction is the context of safety (see introduction).Rules with variables occurring only on the left hand side directly lead to infinite relations, making the rules unsafe which is no problem in the scope of this thesis.

interpretation of the EDBs is given by $\mathcal{A}$ and remains fixed. Initially, all IDBs are interpreted by the empty set. In each stage, a rule ρ of Π is applied, and some tuples of elements of A are added to the interpretation of the IDB occurring in in the head of rule ρ. Formally, for every k-ary IDB R we define a sequence $(R_i^{\Pi,\mathcal{A}})_{i\geq 0}$ of k-ary relations on the universe A of $\mathcal{A}$. We let $R_0^{\Pi,\mathcal{A}} = \emptyset$ for all IDBs R. Suppose now, we have defined $R_{i-1}^{\Pi,\mathcal{A}}$ for all IDBs R. In stage i, we choose a rule ρ in Π, say $P\bar{x} \leftarrow Q_1\bar{y}_1, \ldots, Q_m\bar{y}_m$. An *instantiation* of ρ at stage i consists of tuples $\bar{a}, \bar{b}_1, \ldots, \bar{b}_m$ of elements of A matching the lengths of the variable tuples $\bar{x}, \bar{y}_1, \ldots, \bar{y}_m$, such that

- If two variables of the rule are equal, then the corresponding elements of the tuples are equal as well. For example, if $x_r = y_{st}$, then $a_r = b_{st}$.
- For $1 \leq r \leq m$: If Q_r is an EDB, then $\bar{b}_r \in Q_r^{\mathcal{A}}$. If Q_r is an IDB, then $\bar{b}_r \in Q_{r\,(i-1)}^{\Pi,\mathcal{A}}$.

We let

$$P_i^{\Pi,\mathcal{A}} = P_{i-1}^{\Pi,\mathcal{A}} \cup \{\bar{a} \mid \text{there exist tuples } \bar{b}_1, \ldots, \bar{b}_m \text{ such that } \bar{a}, \bar{b}_1, \ldots, \bar{b}_m \text{ is an instantiation of rule } \rho \text{ at stage } i\}.$$

For all IDBs $R \neq P$, we let $R_i^{\Pi,\mathcal{A}} = R_{i-1}^{\Pi,\mathcal{A}}$. To turn this into a well-defined deterministic process, we cycle through the rules ρ of Π in some fixed order. It can be shown that the result of the computation does not depend on this order. (It will be convenient later to apply only one rule at each stage, that is why set up the computation this way.) The application of a datalog rule to some stage can also be seen as applying an operator, which is then called the *immediate consequence operator* (see, e.g. [AHV95]).

Note that for all IDBs R and for all $i \geq 0$ we have $R_i^{\Pi,\mathcal{A}} \subseteq R_{i+1}^{\Pi,\mathcal{A}}$. The process either reaches a fixed point after finitely many stages, that is, there is an i_0 such that $R_{i_0}^{\Pi,\mathcal{A}} = R_i^{\Pi,\mathcal{A}}$ for all $i \geq i_0$, or it continues forever (recall that $\mathcal{A}$ may be infinite). In both cases, we let $R_\infty^{\Pi,\mathcal{A}} = \bigcup_{i\geq 0} R_i^{\Pi,\mathcal{A}}$. Then the $R_\infty^{\Pi,\mathcal{A}}$ form a fixed point of the computation, that is, further applications of the rules do not increase the relations. This is obvious if a fixed-point is reached in finitely many stages, but also easy to see if the fixed-point is not reached in finitely many stages. The result of the computation is the interpretation of the IDBs in this fixed point.

We usually write R_i^{Π} and R_∞^{Π} instead of $R_i^{\Pi,\mathcal{A}}$ and $R_\infty^{\Pi,\mathcal{A}}$ if $\mathcal{A}$ is clear from the context. Borrowing some terminology from formal grammars, we say that an IDB P can be *derived* (in Π), if there are rule applications leading to $P_\infty^{\Pi,\mathcal{A}} \neq \emptyset$.

See Example 2.1 and Example 2.2 for some examples for these definitions.

Example 2.1 *As structure $\mathcal{G} = (V, E)$, we use a directed graph. Then the following two rule program Π will compute the transitive closure relation $T^{\mathcal{G}} \subseteq (V^{\mathcal{G}})^2$ of $E^{\mathcal{G}}$:*

$$\begin{aligned} \rho_1: &\quad T(x,y) \leftarrow E(x,y). \\ \rho_2: &\quad T(x,y) \leftarrow E(x,z), T(z,y). \end{aligned}$$

Rule ρ_1 is a non-recursive rule, just copying all tuples from E to T. Rule ρ_2 is recursive and contains an (existentially quantified) body variable z. If we cycle through the rules by applying ρ_1 to derive stages $2i-1$ and ρ_2 for stages $2i$ with $i \in \mathbb{N}$, $T^{\Pi,\mathcal{G}}_{2i-1}$ contains all the pairs (v,w) of vertices of the graph, such that there is a path from v to w with length at most i.

Obviously, we could also apply all non-recursive rules at the beginning and then cycle through recursive rules only, which would lead to the same result. But since the asymptotic running time is (in general) not affected by that, we will cycle through all rules in general.

Example 2.2 *Let $\mathcal{G} = (V, E)$ be the directed representation of an undirected graph G, i.e. for all $v, w \in V^{\mathcal{G}}$ with $(v,w) \in E^{\mathcal{G}}$, we also have $(w,v) \in E^{\mathcal{G}}$. Let Π be the following three rule program*

$$\begin{aligned}
\rho_1: &\quad P(x,y) \leftarrow E(x,z),\, E(z,y).\\
\rho_2: &\quad P(x,y) \leftarrow P(x,z),\, P(z,y).\\
\rho_3: &\quad O \leftarrow P(x,y),\, E(y,x).
\end{aligned}$$

Then rules ρ_1 and ρ_2 will define P to contain all pairs (v,w) of vertices with a path with an even number of edges between them. The nullary IDB O is filled with the nullary tuple by rule ρ_3, if there is a cycle with an odd number of edges in $\mathcal{G}$.

This program can be used to solve the problem TWO-COLORABILITY: The graph $\mathcal{G}$ is colorable with two colors if and only if the IDB O is empty: $O^{\Pi,\mathcal{G}}_\infty = \emptyset$. The equivalence of the non-two-colorability and the existence of odd cycles used here is straightforward.

If a program Π is a program over a structure $\mathcal{A}$ with constants and if Π is allowed to use these constants, constants may occur in the head any body atoms. In rule applications the tuple entries (of tuples in EDBs or IDBs) then have to be equal to the corresponding constants.

An immediate consequence of the definition of datalog computations which can be proved by induction over rule applications is the following well known fact (see, e.g., [AHV95]):

Remark 2.3 *Let $\mathcal{A}$ and $\mathcal{B}$ be two structures over the same vocabulary τ. Let Π be a datalog program over $\mathcal{A}$ and let $h : \mathcal{A} \to \mathcal{B}$ be a homomorphism. Then for each IDB P of Π and each tuple $\bar{a}$ over $\mathcal{A}$ with the same arity as P it holds that:*

$$\bar{a} \in P^{\Pi,\mathcal{A}}_\infty \qquad \Longrightarrow \qquad h(\bar{a}) \in P^{\Pi,\mathcal{B}}_\infty$$

A similar result holds for each stage of the computation of Π, if the same rule sequence is applied on $\mathcal{A}$ and $\mathcal{B}$.

The monotonicity of datalog follows from this remark directly by using the canonical embedding homomorphism:

Remark 2.4 *Let $\mathcal{A}$ be a structure and $\mathcal{B}$ a substructure of $\mathcal{A}$. Let Π be a datalog program over $\mathcal{B}$. Then for each IDB P of Π it holds that:*

$$P_\infty^{\Pi,\mathcal{A}} \subseteq P_\infty^{\Pi,\mathcal{B}}$$

For a detailed introduction to datalog, we refer the reader to [AHV95] and [Ull88a; Ull88b].

2.2.2 Program Parameters

For an easier reference, we define the following set of parameters for a datalog program Π:

Parameter	Meaning
m_L	maximal IDB arity (variables in head of program rules)
m_R	maximal number of different variables occurring in a rule
n_R	number of rules
n_I	number of IDB symbols
m_I	maximal number of IDB occurrences in a rule body

All these parameters are bounded from above by the length $n := |\Pi|$ of Π in some standard encoding.

2.2.3 Datalog and Logics

To gain a logical formalism describing datalog, note that datalog is equivalent to the inequality-free positive existential fragment of least fixed point logic. This equivalence is examined more closely in [KV95], where also a connection to infinitary variable logic is established. Infinitary variable logic with a finite set of variables allows the usage of pebble games to show that certain properties are not definable, which can be transferred to non-computability of properties by datalog programs. We will make use of a special version of these pebble games in Chapter 4, where we give a brief definition of these games. Details about the relation of pebble games, infinitary logic and datalog can be found in [KV95].

2.2.4 Datalog and Negation

Enhanced definitions of datalog including negation and their effects have been extensively studied in literature (see, e.g., [AHV95; Ull88a]). Negations can occur in the rule bodies of datalog programs in different places: In front of EDB relations and

in front of IDB relations. While negations in front of IDB symbols require massive changes in the semantics of datalog and, depending on the approach, also syntactic restrictions, negations in front of EDB are easier to handle. Various approaches have been proposed and investigated for both of these negation variants.

As datalog$^\neg$, we define the extension of datalog containing negations in front of EDB atoms. A datalog$^\neg$ rule is a rule of the form $P\bar{x} \leftarrow Q_1\bar{y}_1, \ldots, Q_m\bar{y}_m$, where each $Q_i\bar{y}_i$ is either an IDB or EDB atom or a negated EDB atom, while the rest remains as in the definition of datalog.

The datalog semantics is extended for rules containing negated EDB atoms. We define the instantiation of a datalog$^\neg$ rule over the structure $\mathcal{A}$, which is the only change in the definition, compared to Section 2.2.1.

Let $\rho : P\bar{x} \leftarrow Q_1\bar{y}_1, \ldots, Q_m\bar{y}_m$ be a datalog$^\neg$ rule. Then an instantiation of ρ at stage i consists of tuples $\bar{a}, \bar{b}_1, \ldots, \bar{b}_m$ of elements of A matching the lengths of the variable tuples $\bar{x}, \bar{y}_1, \ldots, \bar{y}_m$, such that

- If two variables of the rule are equal, then the corresponding elements of the tuples are equal as well. For example, if $x_r = y_{st}$, then $a_r = b_{st}$.
- For $1 \leq r \leq m$: If Q_r is an EDB E, then $\bar{b}_r \in E^{\mathcal{A}}$.

  If Q_r is a negated EDB $\neg E$, then $\bar{b}_r \notin E^{\mathcal{A}}$.

  If Q_r is an IDB I, then $\bar{b}_r \in I^{\Pi,\mathcal{A}}_{i-1}$.

Using this change, the rest of the definitions remains as in Section 2.2.1.

In Chapter 5 and Chapter 6 we will show how to simulate datalog$^\neg$ with negation free datalog, for some cases of linear orders, where this is possible. The basic idea is that the negation of an order atom $\neg(x < y)$ can be translated to the disjunction of an order atom $y < x$ and an equality atom $x = y$. For monadic relations, we may simulate the negation of a monadic relation M_1 by adding the complement as second monadic relation, $M_2 := \overline{M_1}$.

If not stated otherwise explicitly, we will always assume that the datalog programs do not contain any negation at all.

2.3 Datalog Problems

For a fixed structure $\mathcal{A}$, we consider two datalog related problems:

Definition 2.5

DATALOG-NONEMPTINESS$(\mathcal{A})$

Instance: *Datalog program* Π *over* $\mathcal{A}$*, IDB symbol* P *of* Π

Question: *Is* $P^{\Pi,\mathcal{A}}_\infty \neq \emptyset$*?*

Definition 2.6

DATALOG-TUPLE$(\mathcal{A})$

Instance: *Datalog program* Π *over* $\mathcal{A}$*, IDB symbol* P *of* Π *of arity* k*, tuple* $\bar{a} \in A^k$

Question: *Is* $\bar{a} \in P_\infty^{\Pi,\mathcal{A}}$*?*

2.4 Linear Orders and Successor Structures

A *linearly ordered set* is a structure $\mathcal{A} = (A, <^{\mathcal{A}})$ of vocabulary $\{<\}$, where the binary relation $<^{\mathcal{A}}$ is a linear order of the universe A, i.e. a transitive, total, and antireflexive relation. For brevity, we refer to linearly ordered sets just as *linear orders*. Moreover, we usually omit the superscript in $<^{\mathcal{A}}$ and use the symbol $<$ to denote both the relation $<^{\mathcal{A}}$ and the relation symbol $<$. We write $a \leq b$ instead of ($a < b$ or $a = b$).

Let $(A, <)$ be a linear order. $(A, <)$ is *dense without endpoints*, if for all $a \in A$ there are $b, c \in A$ such that $b < a < c$, and for all $a, b \in A$ with $a < b$ there is a $c \in A$ such that $a < c < b$. Contrasting other orders, dense orders allow *quantifier elimination*, which states, that for each first order formula over a dense linear order without endpoints, we can find an equivalent quantifier free formula (see, e.g., [Hod97]).

We now define a distance d on $(A, <)$. Let $a, b \in A$ be two elements from A satisfying $a < b$ and let $C \subseteq A$ the set of elements between a and b, i.e. $c \in C$ if and only if $a < c < b$.

Then the *distance* $d(a, b)$ between two elements $a, b \in A$ with $a \neq b$ is defined as:

$$d(a,b) = \begin{cases} \infty, & \text{if } |C| = \infty \\ |C| + 1, & \text{if } |C| < \infty \end{cases}$$

Additionally, we let $d(a, a) = 0$ for all elements $a \in A$.

We may also consider orders having a minimum or maximum, which will be denoted by *min* and *max*. But also orders with both minimum and maximum may be infinite, as for example the closed interval $[0, 1]$ of real numbers. If not stated otherwise, the constants min and max will not be available to the datalog programs.

We consider linear orders in the strict sense, that is, a linear order is always antireflexive. For orders in the sense of "less-than-or-equal-to", the DATALOG-NON-EMPTINESS problem is trivial, see Section 5.2.1.

If not explicitly defined otherwise, all orders in this thesis will be strict linear orders.

A *successor structure* is a structure $\mathcal{B} = (B, S^{\mathcal{B}}, N^{\mathcal{B}})$, where B is either finite or countably infinite, and for some enumeration $b_0, b_1, \ldots$ of B, the binary relation $S^{\mathcal{B}}$ consists of all pairs (b_i, b_{i+1}), and the unary relation $N^{\mathcal{B}}$ only contains the element b_0.

2.5 Representation of Infinite Structures

For computations over infinite structures special care has to be taken on the representation of these structures. Decidability and complexity results are only meaningful, if the operations on the infinite structures involved are recursive or even within some complexity bounds. Our general focus on programs on fixed infinite structures helps to reduce this problem: If the containment of a tuple in an EDB relation is decidable and independent of the choice of elements, only the time for writing the representation of the tuple entries has to be considered, while the computation of the EDB atoms occurs as a constant in our algorithms and will cause no trouble. If, on the other hand, the running time for a query to the structure depends on the elements involved, say the length of the representation, our complexity results will be subject to a strong influence of the representation of the underlying structure.

2.5.1 Representations and Datalog

For the two problems considered in this thesis, DATALOG-NONEMPTINESS and DATALOG-TUPLE, different requirements on the underlying structure are necessary.

DATALOG-NONEMPTINESS

As discussed in later chapters in detail, our approaches to DATALOG-NONEMPTINESS show that for each structure $\mathcal{A}$ considered, there are only finitely many properties of $\mathcal{A}$ needed to solve DATALOG-NONEMPTINESS($\mathcal{A}$) and these properties are hardwired into an algorithm solving DATALOG-NONEMPTINESS($\mathcal{A}$). This makes our solution completely independent of the representation of $\mathcal{A}$. Only the datalog program and the IDB as input have an effect on the running time of the algorithms solving DATALOG-NONEMPTINESS.

DATALOG-TUPLE

For an infinite structure $\mathcal{A}$, DATALOG-TUPLE($\mathcal{A}$) bears some difficulties with regards to the representation of the input tuple and the accessibility of the structure. To deal with the first difficulty, whenever we consider the DATALOG-TUPLE($\mathcal{A}$) we assume that the universe of the structure $\mathcal{A}$ is a decidable set of strings over some finite alphabet.

To be able to determine the complexity of DATALOG-TUPLE($\mathcal{A}$) exactly, a close look on the operations occurring in our algorithms is necessary. For reasons which are discussed in Chapter 6 and 7, we only investigate DATALOG-TUPLE($\mathcal{A}$) on linear orders $\mathcal{A}$, allowing constants and dense orders as special cases.

To solve DATALOG-TUPLE($\mathcal{A}$) on linear orders $\mathcal{A} = (A, <^{\mathcal{A}})$, we need two types of queries to the underlying structure $\mathcal{A}$, for all elements $a, b \in A$:

- Is $a < b$ satisfied?

- Assume we know $a < b$. Let k be a nonnegative integer. Are there $a_1, \ldots, a_k \in A$ with $a <^{\mathcal{A}} a_1 <^{\mathcal{A}} \cdots <^{\mathcal{A}} a_k <^{\mathcal{A}} b$?

While the first of these two queries is always needed for DATALOG-TUPLE($\mathcal{A}$) on linear orders $\mathcal{A}$, the second query is not needed if $\mathcal{A}$ is a dense order. With the notation introduced in Chapter 5, we may abbreviate the second query as $a <_{k+1} b$.

For both queries we may assume that they are decidable, otherwise it is likely that DATALOG-TUPLE($\mathcal{A}$) would also be undecidable. Since the two queries are used by the solution algorithms for DATALOG-TUPLE($\mathcal{A}$), the evaluation time of these queries plays a role when analyzing the complexity of DATALOG-TUPLE. To put this in a formal context, we will use different algorithms which decide these queries in certain time bounds. For all of these algorithms, the query is first written on a special input tape, then the algorithm is called an delivers the answer ("yes" or "no") using the corresponding resources. The queries will only contain the elements of the input tuple $\bar{a}$ of the DATALOG-TUPLE($\mathcal{A}$) instance $(\Pi, P, \bar{a})$, or some of the finitely many constants (if present) and a value for k.

Definition 2.7 *Let $\mathcal{A} = (A, <)$ be a linear order without constants and let $a, b \in A$, k be a nonnegative integer. An* order query *is a query*

$$q^{\mathcal{A}}_{<}(a, b) := a < b?$$

It has answer "yes", if $a <^{\mathcal{A}} b$, and "no" otherwise. A distance query *is a query*

$$q^{\mathcal{A}}_{d}(k, a, b) := a <_k b?$$

It has answer "yes", if there are $a_1, \ldots, a_{k-1} \in A$ satisfying $a <^{\mathcal{A}} a_1 <^{\mathcal{A}} \cdots <^{\mathcal{A}} a_{k-1} <^{\mathcal{A}} b$. For a linear order with finitely many constants $\mathcal{A} = (A, <, c_1, \ldots, c_\ell)$, we allow constants to appear instead of either a or b in these two queries, with the obvious meaning.

Definition 2.8 *A* structure oracle *is an algorithm getting as input an order query or a distance query and which outputs the answer of the query in constant time.*

Besides this rather abstract approach, we will have a look at two concrete implementations of infinite orders and how the given implementation influences our results. As example for a dense order, we consider the rational numbers and for non-dense orders, we use the example of the ordered integers, a discrete order. We start with the easier, discrete example.

Representation of the Ordered Integers

The discrete order $\mathcal{Z} = (\mathbb{Z}, <)$ of the integers can be represented over a finite alphabet $\Sigma = \{0, 1, \#\}$: Each integer x can be uniquely represented as finite string from Σ^* using 0 or 1 as first character for the sign of x, followed by a # and the binary representation of the absolute value of x *without leading zeros*. Each number x can

be represented using $\Theta(\log(x))$ characters and tuple entries can be separated by a "##". For this representation, it can be checked in linear time, if the input denotes an integer number or a tuple of integer numbers. For the algorithms presented later, we also need to solve the two queries already introduced:

Order query $q_<^{\mathcal{Z}}(a,b)$: To check this, we first have a look at the signs of the two numbers, since for different signs of a and b, the answer is immediate. For coinciding signs, further checks are needed, where for a and b both negative, simply the argument order is reversed. A comparison of the lengths of a and b is possible in linear time, and different lengths immediately lead to a result, since a longer representation denotes bigger absolute value (by our exclusion of leading zeros). If a and b have the same length l we compare the representation of the absolute values of the numbers character-wise, starting with the most significant one and iterating to the lowest one and derive an answer to the query in at most linear time (on a multi-tape Turing machine, polynomial time on a single tape Turing machine).

Distance query $q_d^{\mathcal{Z}}(k,a,b)$: First, we carry out the check $q_<^{\mathcal{Z}}(a,b)$ as shown. If this test succeeds, we calculate $c := b - a + 1$, which can be done in linear time and is essentially one run over the representations of a and b, with details depending on the sign of a and b. The result of the query is then the same as of the query $q_<^{\mathcal{Z}}(k,c)$, which can be calculated in linear time.

Representation of the Ordered Rationals

The rationals $\mathcal{Q} = (\mathbb{Q}, <)$ can be represented over the finite alphabet $\Sigma = \{0, 1, \#\}$ in a similar way: The representation of each rational $q \in \mathbb{Q}$ consists of a 0 or 1 for the sign, a delimiter #, followed by the binary representation of the numerator n, followed by #, and the binary representation of the denominator d. To ensure a unique representation, we choose the denominator and numerator to be coprime[4] positive integers and we choose their binary representations without leading zeros.

Whether a string is a valid representation can easily be checked in quadratic time, by first checking if the encodings of the parts n and d and the sign are valid and then checking the coprimality of n and d by using Euclid's algorithm (which has quadratic running time, see, e.g., [Sch01]).

Since $\mathcal{Q} = (\mathbb{Q}, <)$ is a dense order, we will only need one of the two queries to be answered:

Order query $q_<^{\mathcal{Z}}(a,b)$: To check this, we first have a look at the signs of the two numbers. For different signs of a and b, the answer is immediate. For coinciding signs, further checks are needed, where for negative a and b, the argument order is reversed.

[4] Two natural numbers are coprime if they have no common divisor greater than 1.

Without loss of generality assume that a and b are positive and are represented as $a = n_a/d_a$ and $b = n_b/d_b$. We may then check if $n_a \cdot d_b < n_b \cdot d_a$, which is equivalent to $a < b$. The computation of the two products and the inequality test can be carried out in quadratic time altogether.

2.5.2 Computational Model Theory

We will give a very brief survey of known representations of infinite structures which closely follows the nice presentation in [BG04]. Computational model theory extends finite model theory and its methods to infinite structures, representing each domain in a finite way and providing an effective semantics, i.e. the operations used on these structures have to be decidable. Besides these minimal requirements, one often demands closure (relations defined by formulae are computable using the representation of the structure) and effective query evaluation (computing a representation of the satisfying set of a formula).

Recursive structures: Recursive structures are the subject of *recursive model theory* (e.g. see [EGNR98]) and *effective model theory* (e.g. see [Chi90; AK00]) and are countable structures, on which the domain and all relations are computable. This approach is too general for most applications, only allowing effective evaluation algorithms for simple formulae.

Constraint databases: Constraint databases (see e.g. [KG94; KLP00]) use quantifier free formula over a fixed infinite background structure (e.g. linear orders) to define relations. The set defined by a such a quantifier free formula is called a *generalized tuple* and as an analogon to finite databases, a finite set of generalized tuples form a *generalized relation*. Constraint databases on different background structures have been focus of research, for example in the survey [Rev95] an ordered structure with additional distance information has been studied: So called gap order constraints are order atoms $x < y$ together with a minimum distance the elements x and y have to satisfy, or together with a maximum distance. Our distance type concept introduced in Chapter 5 utilizes some similar methods to describe the IDB relations of database programs on orders. One of our results derived with theses tools is the uniform boundedness of datalog on linear orders, a more general result than the decidability of datalog on gap orders in [Rev93].

The survey [Rev95] includes a brief overview of other results on the data complexity of datalog on constraint databases with different sets of constraints, e.g. databases on the dense order (augmented with equality atoms) have PTIME-complete data complexity as in the finite case (see [Tom95]).

The input in these cases is a finite encoding of the generalized tuples of the generalized relations of the database and the program is considered fixed. Our results can be seen as an extension of these results to combined complexity,

as we consider the program over the infinite (background) structure as input and may include the finite description of the constraint database as part of our program.

Metafinite structures: Metafinite structures (see e.g. [GG98]) consist of a finite structure (which can be finitely encoded) and an infinite background structure and some functions between them. Usually, the infinite background structure is a fixed structure, which is used as a domain of numerical objects, but with some limitations, e.g. on quantification. A typical example are finite graphs with real valued edge weights. For standard scenarios we could use this kind of structures, for example for the ordered rational numbers together with some constants, but the main focus for metafinite structures are only finite structures with an infinite background.

Tree-interpretable structures: Tree-interpretable structures are structures that are interpretable in the infinite tree $\mathcal{T}^2 = (\{0,1\}^*, \sigma_0, \sigma_1)$ via a one-dimensional monadic second-order interpretation (see [BG04] for details), e.g. context free graphs, which are configuration graphs of push-down automata (see [MS83; MS85]), HR-equational and VR-equational graphs (see [Cou89]), prefix-recognizable graphs (see [Cau96]). In Chapter 7 we discuss why our solution methods only derive a uniform bound for DATALOG-NONEMPTINESS($\mathcal{A}$) on tree orders, and also in this chapter it is shown that for this problem the solution methods are independent of the representation of the structure.

Automatic structures: Using finite automata to describe structures is the approach of automatic structures, which are presented in [BG00; BG04]. The universe has to be a regular language over some finite alphabet. The equality relation and all relations of the given structure have to be recognizable by finite automata (reading their input words simultaneously). This easy verification of relation membership seems to make them ideal for the DATALOG-TUPLE, but one has to make sure that the order and distance queries we use are expressible and computable within reasonable complexity bounds. On ω-*automatic* structures (using infinite words and Büchi automata instead of finite automate) also the queries needed for colored orders are decidable, since on this class of structures first order logic extended by the quantifier "there are infinitely many" is decidable. On the other hand, not all structures are automatic (or ω-automatic) which limits this approach.

In fact, the representation of the ordered integers in Section 2.5.1 can be implemented as automatic structure and also the distance query $q_d^{\mathcal{Z}}(k,a,b)$ can be decided by a finite automaton (FA):

- The encoding of the integers as proposed with a sign bit, the delimiter # and the binary representation of the absolute value without leading zeros can clearly be decided by a FA.

- Whether $a < b$ holds, i.e. if the tuple (a, b) is contained in the order relation $<$, can be decided by a FA reading the representations of a and b simultaneously: The first character determines the signs of a and b. For $b < 0$ and $a > 0$ the FA rejects, while for $a < 0$ and $b > 0$ the FA accepts, and in both cases the rest of the input is irrelevant. For coinciding signs, the rest of the input is scanned and compared bitwise, from the lowest bit to the highest bit. Using two different states, the FA remembers whether the numbers a' and b' corresponding to the bits read so far satisfy $a' < b'$ or not. When the next bit is read, the state may only change if the next bits of a and b are different. If the two representation strings of a and b have the same length, this state information will determine if the FA accepts. If the strings are of different length, the FA accepts if the representation of b is longer than the representation of a.

- The distance query $q_d^{\mathbb{Z}}(k, a, b)$ is a ternary relation and can be decided by a FA reading the representations of k, a and b simultaneously, where the representation of k also includes a sign bit for an easier description, although k is always nonnegative. The FA first reads the sign bit of a and b (and also k) and then the delimiter #. Now the FA computes several tasks simultaneously, where we sketch the case of $a > 0$ and $b > 0$:

  - The FA checks if $a < b$ is satisfied, as above, and
  - the FA adds a and k bitwise storing the carry bit using different states, and
  - the FA checks if $a + k < b$, also bitwise.

  The check for $a + k < b$ can be carried out bitwise by adding the current carry bit (stored in some state) and the current bits of a and k and performing an comparison of the result with the current bit of b. The cases where the representations of $a + k$ and b are of different length are handled as described above for $a < b$. For $a < 0$ and $b < 0$ the roles of a and b are simply reversed in the FA. For $a < 0$ and $b > 0$, the check $a + k < b$ is equivalent to $k < b + (-a) = b + |a|$, which can be carried out using bitwise addition and comparison of the representations of k, b and a.

2.6 Allen's Interval Algebra

Ordered structures play an important role in temporal reasoning, where orders or intervals with endpoints in a linear order are used to model temporal data and dependencies. In the related field of spatial reasoning, intervals are often replaced by higher dimensional sets (see e.g. [PSV96]), where the set relations cannot easily be replaced by order formulae over interval endpoints as in the one-dimensional case, why we focus on temporal reasoning. We do not understand temporal reasoning as

temporal logic with an additional set of temporal quantifiers, but as the mere storage of temporal data and manipulation, e.g. in an algebra of intervals.

As an example for the connection of our datalog results with temporal reasoning, we will have a closer look at Allen's interval algebra and show how to solve datalog programs over this algebra using datalog programs over linear orders.

Allen's interval algebra, introduced in [All82], is an algebra of relations over open intervals on the real line. These interval relations are built as unions from the 13 basic relations describing the pairwise relative end points of two intervals (x^-, x^+) and (y^-, y^+) as shown in the table taken from [KJJ04] (see Table 2.1, which includes an illustrative example for the intervals x and y in each case, their alignment shown with strings $xxxx$ and $yyyy$). The algebra of these 2^{13} relations is equipped with the operations *converse* (denoted by $\cdot^{-1}$), *intersection* $\cap$ and *composition* $\circ$.

The complexity of constraint satisfaction problems over Allen's interval algebra and variants has been extensively studied (see, e.g., [KJJ04; Mei96; NB95]). While constraint satisfaction problems may be viewed as DATALOG-NONEMPTINESS for programs with a single rule, here we are interested in the complexity of full datalog over the interval algebra. On the other hand, our results about constraint satisfaction problems on linear orders may be transferred to Allen's interval algebra in a similar way as for datalog, as presented in this section.

Table 2.1: The 13 basic relations of Allen's interval algebra. The obvious inequalities $x^- < x^+$ and $y^- < y^+$ of each case have been omitted.

Basic relation		Converse relation		Example	Endpoints
x precedes y	$\mathbf{p}$	y preceded by x	$\mathbf{p}^{-1}$	xxx yyy	$x^+ < y^-$
x meets y	$\mathbf{m}$	y met by x	$\mathbf{m}^{-1}$	$xxxx$ $yyyy$	$x^+ = y^-$
x overlaps y	$\mathbf{o}$	y overlapped by x	$\mathbf{o}^{-1}$	$xxxx$ $yyyy$	$x^- < y^- < x^+ < y^+$
x during y	$\mathbf{d}$	y includes x	$\mathbf{d}^{-1}$	xxx $yyyyyyy$	$y^- < x^-$, $x^+ < y^+$
x starts y	$\mathbf{s}$	y started by x	$\mathbf{s}^{-1}$	xxx $yyyyyyyy$	$x^- = y^-$, $x^+ < y^+$
x finishes y	$\mathbf{f}$	y finished by x	$\mathbf{f}^{-1}$	xxx $yyyyyyyy$	$y^- < x^-$, $x^+ = y^+$
x equals y	$\equiv$			$xxxxx$ $yyyyy$	$x^- = y^-$, $x^+ = y^+$

Let $\mathcal{I}$ denote the structure whose universe consists of all open intervals on the real line, and whose relations are the relations of the interval algebra. We observe that datalog programs over $\mathcal{I}$ can easily be translated into programs over the linear order $(\mathbb{R}, <)$ and vice versa:

Lemma 2.9 *DATALOG-NONEMPTINESS($\mathcal{I}$) is LOGSPACE-equivalent to DATALOG-NONEMPTINESS($\mathcal{R}$), where $\mathcal{R} = (\mathbb{R}, <)$.*

Proof: The reduction from DATALOG-NONEMPTINESS($\mathcal{I}$) to the DATALOG-NONEMPTINESS($\mathcal{R}$) is straightforward by replacing the interval variables by endpoint variables, according to the definition in Table 2.1. For example, two interval

variables x and y would be replaced by x^-, x^+, y^- and y^+ and for consistency the atoms $x^- < x^+$ and $y^- < y^+$ would be added. An atom $\mathbf{p}(x, y)$ would then be replaced by $x^+ < y^-$.

Since we do not allow any equality relation to be used, we simulate equality by variable reusing, e.g. we add an additional rule

$$EQ(x^-, x^+, x^-, x^+) \leftarrow .$$

and then translate "$\equiv (x, y)$" to $\mathrm{EQ}(x^-, x^+, y^-, y^+)$ forcing the endpoints of x and y to be equal.

For the other direction, we transform the program Π over $(\mathbb{R}, <)$ to Π' by replacing all atoms $x < y$ by $\mathbf{p}(x, y)$. Then Π' is satisfiable if and only if Π is satisfiable: If $\mathbf{p}(x, y)$ holds, then $x^- < y^-$ is satisfied. If on the other hand $x < y$ holds, then there are elements x^+ and y^+ such that $\mathbf{p}((x, x^+), (y, y^+))$ is satisfied, because the order $(\mathbb{R}, <)$ is dense.

Both reductions can clearly be carried out in logarithmic space. □

If we allow our datalog programs direct access to the interval endpoints of given intervals in Allen's interval algebra, this construction can also be used to show the equivalence of the DATALOG-TUPLE on both structures. In this case, the lower endpoints of the intervals in the resulting IDB relations of Π' would give a tuple of the IDB relations of Π, and the tuples in the IDB relations of Π' could be used to create the solutions of Π.

Some recent extensions of Allen's interval algebra could be used to gain stronger results: In [KJJ04] the algebra was extended by length constraints on intervals. Using one of the relations, the **meets**-relation $\mathbf{m}$, and a simple length constraint, a successor structure can be defined:

$$S(x, y) \leftarrow \mathbf{m}(x, y),\ \mathrm{length}(x) = \mathrm{length}(y) = 1.$$

Then S acts as a successor relation. Using a marked element as zero value, this gives a successor structure increasing the complexity of datalog problems, considered here, to non-decidable, as shown in Chapter 4.

Chapter 3

Lower Bounds for Datalog

Outline of this Chapter

In this chapter, we first introduce one of the main tools of this thesis: A Turing machine simulation by datalog programs. Our first application of this simulation is the proof of some lower bounds for datalog, including a lower bound for datalog on orders.

3.1 Universal Turing Machine Simulation

The hardness results in this section are either from [DEGV97], or they can fairly easily be proved by the techniques used in [DEGV97], and this survey summarizes some known facts about datalog: For finite structures $\mathcal{A}$, DATALOG-NONEMPTINESS($\mathcal{A}$) is in EXPTIME. For every structure $\mathcal{A}$ whose universe contains at most one element, DATALOG-NONEMPTINESS($\mathcal{A}$) is in PTIME. Conversely, for every structure $\mathcal{A}$, DATALOG-NONEMPTINESS($\mathcal{A}$) is PTIME-hard, because the satisfiability problem for propositional Horn clauses is equivalent to the nonemptiness problem for datalog programs with only 0-ary relation symbols.

As soon as a structure contains two distinguishable elements, the nonemptiness problem becomes EXPTIME-hard. This will be made precise in Lemma 3.2 below. For the readers convenience, we sketch the proof which requires some preparation.

Assume, that in some structure $\mathcal{A}$ with universe A, we can *define* a successor structure. This means that there exists a datalog program Π with an m-ary IDB U, a $2m$-ary IDB S, and an m-ary IDB N such that the structure $\mathcal{B} = (B, S^{\mathcal{B}}, N^{\mathcal{B}})$ with $B = U_\infty^{\Pi,\mathcal{A}}$, $S^{\mathcal{B}} = S_\infty^{\Pi,\mathcal{A}}$, and $N^{\mathcal{B}} = N_\infty^{\Pi,\mathcal{A}}$ is a successor structure. Then a given Turing machine transition function can be translated to a datalog program defining the following IDB relations:

symbol$_\sigma(\bar{x}, \bar{y})$**:** In step $\bar{x}$ of the computation the tape cell $\bar{y}$ contains the symbol σ (one IDB for each symbol $\sigma \in \Sigma$ of the Turing machine tape alphabet Σ).

cursor$(\bar{x}, \bar{y})$**:** At step $\bar{x}$ the cursor points to cell $\bar{y}$.

state$_s(\bar{x})$**:** In step $\bar{x}$ the Turing machine is in state s (one IDB for each state s of the Turing machine).

accept: The computation has reached an accepting state.

Here $\bar{x}$ and $\bar{y}$ range over elements of the defined successor structure $\mathcal{B}$ and hence can be viewed as encoding natural numbers, which are used to address time steps and tape cells. We may define auxiliary IDB relations ensuring the consistency of the simulation and encoding the input on the tape of the machine. The main part of our Turing machine simulation is the translation of the transition function. If the transition function is given as a table, each row will be translated to three datalog program rules:

Assume we have the transition $\delta(s, \sigma) = (s', \sigma', \leftarrow)$, where s is the old state, s' the new one, σ the tape symbol read and σ' the tape symbol written, after which a left movement ($\leftarrow$) of the cursor is done. Then we encode this by the following rules:

$$\begin{aligned}
\mathbf{symbol}_{\sigma'}(\bar{x}', \bar{y}) &\leftarrow \mathbf{state}_s(\bar{x}), \mathbf{symbol}_\sigma(\bar{x}, \bar{y}), \mathbf{cursor}(\bar{x}, \bar{y}), \\
&\quad S(\bar{x}, \bar{x}'). \\
\mathbf{cursor}(\bar{x}', \bar{y}') &\leftarrow \mathbf{state}_s(\bar{x}), \mathbf{symbol}_\sigma(\bar{x}, \bar{y}), \mathbf{cursor}(\bar{x}, \bar{y}), \\
&\quad S(\bar{x}, \bar{x}'), S(\bar{y}', \bar{y}). \\
\mathbf{state}_{s'} &\leftarrow \mathbf{state}_s(\bar{x}), \mathbf{symbol}_\sigma(\bar{x}, \bar{y}), \mathbf{cursor}(\bar{x}, \bar{y}), \\
&\quad S(\bar{x}, \bar{x}').
\end{aligned}$$

Some additional inertia rules assure, that the tape contents does not change at other positions than the cursor position, and some initialization rules define a valid starting configuration, with a tape containing the input padded with spaces at time step 0.

Then we have a program, computable in logarithmic space from the machine description, whose IDB *accept* is derivable if and only if a machine run accepts in a number of steps bounded by the size of the successor structure. The arity of the IDBs introduced is at most $2m$ and the program size is the size of the Turing machine description, increased by factor m (and some constant).

This simulation does not only work for the question, whether an IDB can be derived from the input facts, but also for both of the problems given in section 2.2: If the Turing machine accepts, there has to be a derivation of the IDB **accept**. Since **accept** has no variables, i.e. arity 0, its IDB relation can only contain the empty tuple or be the empty relation. The first case happens, if **accept** is derivable, while the second occurs, if no successful run is possible. Hence the following are equivalent, where () denotes the 0-ary tuple:

- **accept** can be derived in a run of Π

- $\mathbf{accept}^{\Pi}_{\infty} \neq \emptyset$
- $() \in \mathbf{accept}^{\Pi}_{\infty}$.

If the notation of the 0-ary is not desired, the following rule can be used to define an IDB $\mathbf{accept}'(X)$ with the property that **accept** is derivable in a run of Π if and only if $(\mathbf{accept}')^{\Pi}_{\infty} = A$:

$$\mathbf{accept}'(X) \leftarrow \mathbf{accept}.$$

Remark 3.1 ***Multi-tape Machines***
*This simulation can easily be extended to a simulation for a multi-tape Turing machine. For a machine with k tapes, we replace each IDB $\boldsymbol{symbol}_\sigma$ by k IDBs $\boldsymbol{symbol}^i_\sigma$ with $i = 1, \ldots, k$ and the IDB **cursor** by k IDBs $\boldsymbol{cursor}^i$ for $i = 1. \ldots, k$. These IDBs describe the symbols and cursor positions on the k different tapes. The datalog rules are again created from the transition function and each rule contains the information about the current cursor position and tape contents for all k tapes.*

In the following sections and chapters this simulation will be used for successor structures of different sizes. Finite successor structures will lead to the completeness bounds in the next section, ranging to countable infinite successor structures to show undecidability in the next chapter.

3.2 EXPTIME-hard Lower Bound

Generalizing an idea from [DEGV97], we assume, that we have two disjoint sets which we may use to define a successor structure of exponential size

Lemma 3.2 *Given a structure $\mathcal{A}$ such that two relations $R_0, R_1 \subset \mathcal{A}^k$, $k \in \mathbb{N}$, can be defined by a datalog program on $\mathcal{A}$, such that*

$$R_0 \cap R_1 = \emptyset,\ R_0 \neq \emptyset,\ R_1 \neq \emptyset.$$

Then DATALOG-NONEMPTINESS($\mathcal{A}$) is EXPTIME-hard.

Proof: Without loss of generality we may assume that $\mathcal{A}$ actually contains two k-ary relations R_0, R_1 which are nonempty and disjoint. Hence we can use these relations as EDB predicates in a datalog program. We prove that any deterministic Turing machine computation on input x, with $|x| = n$ and time bound $t(x) = 2^m$ ($m = m(n)$ being a function with variable n) can be simulated by a datalog program with IDBs having arity at most $2 \cdot k \cdot m$.

The elements in R_0 are used as 0 and the elements in R_1 as 1 to build a successor structure of binary vectors of arity m, leading to a successor structure over $[0..2^m - 1]$. For $i = 1, \ldots, m$, we inductively define relations U^i, S^i, F^i, L^i with datalog rules,

where $U^i \subset \mathcal{A}^m$ is the universe and S^i a successor relation on U^i, and F^i contains the first (smallest), L^i the last (biggest) element of U^i.

We start with $U^1 = R_0 \cup R_1$, $S^1 = R_0 \times R_1$, $F^1 = R_0$ and $L^1 = R_1$. By induction over i, our datalog program defines

$$\begin{aligned} U^{i+1} =& U^1 \times U^i, \quad F^{i+1} = F^1 \times F^i, \quad L^{i+1} = L^1 \times L^i \\ S^{i+1} =& \left\{ (z\bar{x}, z\bar{y}) \;\middle|\; (\bar{x}, \bar{y}) \in S^i \wedge z \in U^1 \right\} \\ & \cup \left\{ (z\bar{x}, z'\bar{y}) \;\middle|\; (z, z') \in S^1 \;\wedge\; \bar{x} \in L^i \;\wedge\; \bar{y} \in F^i \right\} \end{aligned}$$

The details can be found in [DEGV97] with slight modifications.

The maximal arity of any IDB relations involved is $2 \cdot k \cdot m$, defining the successor between two m-tuples of entries that have arity k.

By the construction of the Turing machine simulation, any machine computation running at most 2^m steps can be simulated using datalog programs with maximal arity $2 \cdot k \cdot m$, which concludes the proof. □

The first application of this simulation is the case we are interested in most, the case of strict linear orders:

Corollary 3.3 *For every linear order $\mathcal{A} = (A, <)$ with at least two elements, the DATALOG-NONEMPTINESS($\mathcal{A}$) is EXPTIME-hard.*

Proof: Let U_0 be the binary relation $x < y$ and U_1 the converse $x > y$. □

If we allow constant symbols we can easily take two constant symbols interpreted as different constants to define the two disjoint sets needed for the simulation. If we denote the constants by 0 and 1, we have the same situation as in [DEGV97] and are able to use these constant for the initialization rules of the program:

$$\begin{aligned} U_0(0) \;&\leftarrow\; . \\ U_1(1) \;&\leftarrow\; . \end{aligned}$$

Lemma 3.2 shows the EXPTIME-hardness for such structures with two constants having different interpretation.

Example 3.4 *If $\mathcal{A} = (A, <, c)$ is a strict order with only one constant symbol c, we could set*

$$\begin{aligned} U_0(c) \;&\leftarrow\; . \\ U_1(X) \;&\leftarrow\; c < X. \end{aligned}$$

if c is not a maximal element in the order $<$. If c is maximal, then we could use $U_1(x) \leftarrow X < c$ instead. In both cases we get two disjoint relations of arity 1. This example is a generalization of the ordered structure of the natural numbers $(\mathbb{N}, <, 0)$.

3.3 Summary of this Chapter

We have transferred the universal Turing machine simulation introduced in [DEGV97] to structures with two distinguishable sets: If two disjoint sets of some arity can be defined by a datalog program on a structure $\mathcal{A}$, then DATALOG-NONEMPTINESS($\mathcal{A}$) and DATALOG-TUPLE($\mathcal{A}$) are EXPTIME-hard.

As applications of this lemma, we have shown that DATALOG-NONEMPTINESS and DATALOG-TUPLE on strict linear orders are EXPTIME-hard, raising the well known PTIME-hard lower bound, derived from propositional logic programs.

Open problems

We will show a higher upper bound for DATALOG-TUPLE Chapter 5, so maybe there is room for improvement of the lower bounds:

Question 3.1 *Can we show a higher lower bound for DATALOG-TUPLE on orders?*

Of course, this order would have to be a non-dense order, since the tuple problem on dense orders has EXPTIME-complete complexity, as will be shown in Chapter 5.

Chapter 4

Undecidability of Datalog on Infinite Structures

Outline of this Chapter

In this chapter, the Turing machine simulation is used to show the undecidability of DATALOG-NONEMPTINESS and DATALOG-TUPLE on successor structures. This is then generalized to structures, which to some extent allow to simulate a successor structure, exploring the possibilities of positive existential quantification as used for datalog. Following that, we have a short look at the decidability of DATALOG-NONEMPTINESS and boundedness. The last section of this chapter deals with the question, if for all structures DATALOG-NONEMPTINESS is in EXPTIME (as shown in later chapters) or undecidable. We define some infinite structures on which we reach some given lower bound using the Turing machine simulation, and on which we show some (unfortunately exponential higher) upper bound using pebble games. This shows that there is no dichotomy EXPTIME vs. undecidable for datalog on infinite structures.

4.1 Successor Structures

In the previous chapter we have used a Turing machine simulation to show lower complexity bounds on finite successor structures which were defined using datalog programs. In this chapter we will study the power of datalog programs if we use predefined infinite successor structures. For monadic datalog programs on successor structures the decidability results from well-known facts: Each monadic datalog program can be translated to a monadic least fixed point formula (MLFP) which does not make use of negations or universal quantification, i.e. a formula in the positive existential fragment of MLFP. For every monadic relation defined by a LFP formula a MSO formula defining the same relation can easily be constructed, hence also for every monadic datalog program. By the decidability of MSO on successor structures, as shown in [Büc60], the decidability of DATALOG-NONEMPTINESS and DATA-

LOG-TUPLE follows for monadic datalog programs. In the following section we will show the undecidability for datalog programs with IDB arity greater than one on successor structures.

If we allow the datalog programs to have arbitrary IDB arity, an easy transfer of the ideas from [SS82] becomes possible, where partial recursive functions are simulated by logic programs with the only functions being the 0 function and the successor function. Partial recursive functions are a well known universal computing formalism, defined from the zero function $f(x) = 0$, the successor function $f(x) = x + 1$, projection functions like $f(x_1, \ldots, x_n) = x_i$ using composition $f(\bar{g}(\bar{x}))$, primitive recursion and a minimalization operator $f(\bar{x}) = \mu.y(g(\bar{x}, y) = 0)$. All these can be converted to logic programming rules and consequently to datalog programs on a structure with successor relation and zero relation which are the usual successor structures as defined in 4.1. While for each occurrence of the minimalization operator μ, two IDB relations are needed, all other functions can be translated one to one to IDBs, with the datalog rules mimicking the function definitions without overhead. A main disadvantage of this approach is that it is quite natural for DATALOG-TUPLE, which then becomes equivalent to questions like "$f(x) = y$?", but the interpretation of DATALOG-NONEMPTINESS in this case would be the question if a function has a nonempty domain, which is an unnatural view for many problems.

Coming back to the Turing machine simulation introduced in Section 3.1, we observe that on an infinite successor structure $\mathcal{A} = (A, S, N)$, we can simulate any Turing machine computation. As stated in this section (by letting $m = 1$), IDB arity two suffices for this simulation on an infinite successor structure, leading a sharp cut to the decidable monadic case. We specialize this observation by a classic undecidability proof using a halting problem reduction:

Lemma 4.1 *Let $\mathcal{A} = (A, S, N)$ be an infinite successor structure. Then DATALOG-NONEMPTINESS($\mathcal{A}$) and DATALOG-TUPLE($\mathcal{A}$) are undecidable for datalog programs with IDB arity at most two on $\mathcal{A}$.*

Proof: We use the Turing machine simulation from Chapter 3 to simulate a machine M on the empty word as input. This is clearly a reduction of the halting problem with empty input to DATALOG-NONEMPTINESS($\mathcal{A}$) and DATALOG-TUPLE($\mathcal{A}$) which shows the undecidability of both problems, since the IDB **accept** can be derived if and only if M is halting with empty input. □

4.2 Successor-Like Structures

The ideas developed for showing the undecidability of DATALOG-NONEMPTINESS and DATALOG-TUPLE on successor structures can be transferred to structures similar to a successor structure. We will have a look at two scenarios of relaxing successor structures: On infinite trees, an element is allowed to have more than one successor, but there is still only a unique predecessor for each element. This condition

excludes directed or undirected cycles (if we view our structure as directed graph). As second example we have a look at structures which may contain cycles, but appearing in a very restricted manner only.

Infinite Trees

Compared to the method using partial recursive functions as sketched above, the biggest advantage of proof utilizing the Turing machine simulation is, that it also holds for DATALOG-NONEMPTINESS and loses the dependency on the representation of the tuples for DATALOG-TUPLE. So we may extend it to structures where an enumeration or representation of the elements is not so obvious any more. The only thing we need for our simulation is the ability to define a successor substructure to run our simulation on. Since datalog comprises a positive existential fragment of logic without the inequality relation, care must be taken that we can distinguish all elements of the defined successor substructure and distinguish between elements in this substructure and the rest. Having cycles of arbitrary length in our structure renders them useless for the positive existential definition of a successor substructure, since we may not be able to distinguish between different elements, and instead of using an infinite successor structure of pairwise different elements we may end up iterating through a cycle again and again. While we will take advantage of cycles in later sections (see Sections 4.2.2 and 4.3), we will now concentrate on structures or graphs without cycles or with cycles appearing in a very controlled manner.

Corollary 4.2 *DATALOG-NONEMPTINESS($\mathcal{A}$) and DATALOG-TUPLE($\mathcal{A}$) are undecidable for datalog programs with IDB arity at most two on structures $\mathcal{A} = (A, N, S)$, where N is a unary relation and S is a binary relation such that $\mathcal{A}$ forms a directed graph with edge relation S and the following properties:*

- *For all $n, n' \in N$ with $n \neq n'$ we have $S^*(n) \cap S^*(n') = \emptyset$*
- *For all $n \in N$ the substructure $S^*(n)$ is a tree.*
- *There exists an $n \in N$ such that $S^*(n)$ has an infinite path.*

Here S^* denotes the transitive closure of S and the set $S^*(n)$ contains all elements reachable from n by S^*:

$$S^*(n) = \{n' \mid \exists l \geq 0\ \exists x_1, \ldots, x_l : S(n, x_1) \wedge S(x_1, x_2) \wedge \cdots \wedge S(x_l, n')\} \cup \{n\}$$

Proof: The proof of Lemma 4.1 can easily be repeated on such an infinite graph structure $\mathcal{G}$. If there is an accepting Turing machine simulation on a successor structure as above, then the infinite path can be used to derive the same computation, hence also on $\mathcal{G}$ the **accept** IDB can be derived.

If on the other hand, the simulation on the infinite graph structure $\mathcal{G}$ leads to the derivation of the **accept** IDB, then there must be a derivation of $\mathbf{state}_{\text{yes}}(x)$ for

some value a for x. Since for each $c \in \mathcal{G}$ the predecessor of c in $\mathcal{G}$, i.e. the value b satisfying $S(b, c)$, is unique, and since the elements reachable from elements of N are partitioned into disjoint sets, we have a unique $n \in N$ with a unique derivation starting at $N(n)$, deriving $\mathbf{state}_{\text{yes}}(a)$. By definition of the simulation IDBs and by replacing the EDBs of $\mathcal{G}$ by those of the successor substructure from n to a, we get an accepting computation on this successor substructure. □

Successor structures with some cycles

We may even allow cycles if they appear in a limited manner. Let $\mathcal{L} = (\mathbb{N}, L, N)$ be the structure with $N = \{0\}$ and L the successor relation extended by back edges to smaller elements (see Figure 4.1):

$$L = \{(x, x+1)\} \cup \{(y, z) \mid y > z\}$$

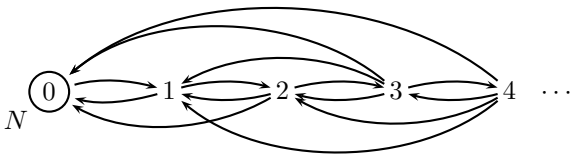


Figure 4.1: Structure $\mathcal{L} = (\mathbb{N}, L, N)$. The arcs show the relation L (up to element 4)

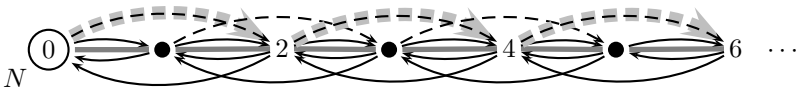
That is, in $\mathcal{L}$ we have the linear successor structure over the nonnegative integers and additionally for each element x infinitely many edges from all bigger elements pointing back to x creating at each element x and for any $k > 1$ a cycle of length k containing x. Obviously, the above Turing machine simulation would not work on this structure, since the successor and predecessor of each element are not well defined. But we may define a successor substructure of $\mathcal{L}$ using datalog:

$$\begin{aligned}
\mathbf{sym}(x, y) &\leftarrow L(x, y), L(y, x).\\
\mathbf{suc}(x, y) &\leftarrow \mathbf{sym}(x, z), \mathbf{sym}(z, y), L(y, x).\\
\mathbf{even}(x) &\leftarrow N(x).\\
\mathbf{even}(x) &\leftarrow \mathbf{even}(y), \mathbf{suc}(y, x).
\end{aligned}$$

A sketch of this structure is given in Figure 4.2.

The IDB **sym** defines an undirected or symmetric version of the successor relation part in L and contains all cycles of length 2. With the help of this we can define the IDB **suc** which contains all cycles of length 3 such that the first and last elements of the cycle are different, which defines exactly the elements which have a distance of 2:

$$\mathbf{suc}^{\Pi}_{\infty} = \{(x, x+2)\}$$



Relations in this figure:

Symbols	Meaning
$0, \bullet, 4, \bullet, 6, \ldots$	$\mathbb{N}$
$0, 2, 4, 6, \ldots$	**even**
(0)	N
⟶	relevant parts of L
═══	**sym**
- - - →	**suc**
▰▰▰▶	part of **suc** used as successor for **even**

Figure 4.2: Structure $\mathcal{L} = (\mathbb{N}, L, N)$ with the IDB relations **sym**. **suc** and **even**.

This relation can be used as a replacement for the successor relation and enables us to use the above Turing machine simulation again, by extending the datalog program by the above rules and making sure that any existential variable is in the scope of the **even** relation defining the universe of a substructure $\mathcal{L}' = (\mathbf{even}, \mathbf{suc}, N)$ of the successor on the even numbers only.

For a successor structure which is augmented with back edges, a similar approach using more variables can be used, if the size of the smallest cycles is constant and known, and if these smallest cycles cover the whole structure and meet at the endpoints of the one back edge in such a cycle, but do not overlap. However, this construction may fail if overlaps are allowed, but may still be possible. If the size of the smallest cycles is unknown or if the cycles are not covering the structure, it is unlikely that a similar approach will succeed. Our approach uses cycles in a constructive way to create a successor relation. Because of the missing negation in datalog it does not seem possible to detect the cycles and use a different approach for parts of the structure where no cycles exist.

On the other hand, if some fixed, finite structure containing cycles exists at the beginning of the successor structure, it is possible to encode this structure into the datalog program creating an extra variable for each of the elements in this part. If this structure is connected to a true successor structure or some cycle structure as above, the program could "read" the first, fixed finite part of the structure, then define its relation N as the first element not in this finite part, define S, and then run on the successor structure starting at N.

4.2.1 Beyond Decidability

Our use of Turing machine simulations to show the undecidability of DATALOG-NONEMPTINESS may lead to the impression, that only recursively enumerable

languages are computable by datalog and that DATALOG-NONEMPTINESS is always recursively enumerable. In our Turing machine simulation, the selected relation for the DATALOG-NONEMPTINESS instance is nonempty whenever the machine accepts, covering the recursively enumerable languages. But with the use of complicated, non recursive EDB relations, also a non recursively enumerable DATALOG-NONEMPTINESS can be created.

4.2.2 Undecidability and Boundedness

During the search for more undecidable cases, one may easily get the impression that unboundedness is a good indicator for the existence of a structure $\mathcal{A}$ with undecidable DATALOG-NONEMPTINESS($\mathcal{A}$). One might come up with the following conjecture:

If on a structure $\mathcal{A}$ there exists an unbounded program Π, i.e. a program for which the fixed point is not reached after finitely many stages, then DATALOG-NONEMPTINESS($\mathcal{A}$) is undecidable.

While for a bounded program the nonemptiness of an IDB relation can be determined by calculating all stages until the fixed point is reached after finitely many stages (using a feasible encoding of the computation results and intermediate relations), this conjecture is the implication the other way round. With this conjecture, boundedness of a program would be equivalent to a decidable DATALOG-NONEMPTINESS.

This link between boundedness and the decidability could help to create other undecidable cases, which do not involve a successor structure, but could allow a more general class of structures, for which DATALOG-NONEMPTINESS is undecidable.

Unfortunately, this conjecture does *not* hold and we will give a counterexample as proof.

Proposition 4.3 *There is an infinite structure $\mathcal{G}_S$, such that there exist unbounded programs on $\mathcal{G}_S$ and DATALOG-NONEMPTINESS($\mathcal{G}_S$) is decidable (even in polynomial time).*

Proof: As structure $\mathcal{G}_S$ we choose a successor structure $\mathcal{G}_S = (U, S, N)$ with universe U, unary relation $N = \{0\}$ and binary successor-like relation

$$S = \{(n, n+1) \mid n \in U\} \cup \{(0, 0)\}$$

which can be viewed as a digraph with edge relation S which consists of a loop and an infinite path starting at this loop, see figure 4.3.

Sub-claim 1: There exists an unbounded program Π on $\mathcal{G}_S$

> We let Π be the following program computing the binary connectivity relation $C = \{(u, v) \in U^2 \mid \text{there exists a path from } u \text{ to } v\}$.

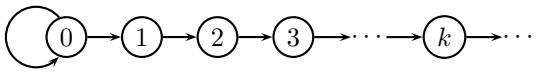


Figure 4.3: The structure $\mathcal{G}_S$ as directed graph.

$$\begin{aligned} C(x,y) &\leftarrow S(x,y). \\ C(x,y) &\leftarrow C(x,z), C(z,y). \end{aligned}$$

Then this program computes all pairs of connected vertices, by

$$C_k^{\Pi} = \{(u,v) \in U^2 \mid \text{there exists a path with at most } 2^{k-1} \text{ internal vertices from } u \text{ to } v\},$$

leading to $C_\infty^\Pi = C$. If the program was bounded, then there would be a $d > 0$ with $C_d^\Pi = C_\infty^\Pi$, but C_d^Π does not contain pairs of vertices with distance 2^d and above, a contradiction. So Π is unbounded.

Sub-claim 2: DATALOG-NONEMPTINESS($\mathcal{G}_S$) is decidable.

We actually show that DATALOG-NONEMPTINESS($\mathcal{G}_S$) can be decided even in polynomial time. Let (Π, P) be a datalog program and P an IDB symbol of Π. By ignoring the variable vectors, we transform Π to a program Π' (with IDB P' corresponding to P), which can be solved in polynomial time as a propositional logic program. It remains to show that $P_\infty^\Pi \neq \emptyset$ if and only if P' can be derived from EDB atoms by program Π.

One direction can be shown easily: If P' cannot be derived, then there cannot be a derivation of P in Π, no matter which variable assignments are used, hence $P_\infty^\Pi = \emptyset$.

For the other direction assume that there is a derivation of P'. Then the EDB atoms in this sequence of rules are all of the form $S(x,y)$, $S(x,x)$ or $N(x)$. Since $\mathcal{G}_S$ contains the vertex 0 with a loop, we may assign the vertex 0 as value to all variables in the derivation, satisfying each EDB atom. Hence, the tuple $a = (0,\ldots,0)$ with the same arity as P will be in P_∞^Π after this evaluation.

□

4.3 Structures for Complexity Functions

While the results in this chapter show the undecidability of the DATALOG-NONEMPTINESS($\mathcal{A}$) for several cases of structures $\mathcal{A}$, Chapters 5, 6 and 7 show singly

exponential complexity for all other cases of structures considered in this thesis – with constants and without. This may lead to the following idea:

Conjecture 4.4 ***(Dichotomy of the Nonemptiness Problem)***
On infinite structures $\mathcal{A}$ without constants, DATALOG-NONEMPTINESS($\mathcal{A}$) is either solvable in EXPTIME or is undecidable.

In the rest of this chapter we shall show, that this conjecture does not hold by creating a set of counter examples.

4.3.1 Lower Bounds

We start by defining a structure, on which DATALOG-NONEMPTINESS will be hard for $TIME(f)$, for a given computable function $f : \mathbb{N} \to \mathbb{N}$.

Definition 4.5 *For each computable function $f : \mathbb{N} \mapsto \mathbb{N}$, the structure $\mathcal{F}^f = (U^f, K^f, S^f)$ with infinite universe U^f consists of two binary relations K^f and S^f and is a disjoint union of finite substructures T_i^f, $i > 0$. We will use the shorthand notations U, K, S for U^f, K^f, and S^f when no confusion is possible.*

The substructure T_i^f consists of the two parts S_i^f and K_i^f, where K_i^f is a complete graph on some nodes $\{v_1^i, \ldots, v_i^i\}$ and edge relation K

$$1 \leq j \neq j' \leq i \quad \Rightarrow (v_j, v_{j'}) \in K,$$

while S_i^f is a successor structure on some elements $\{s_1^i, \ldots, s_{f(i)}^i\}$,

$$\begin{aligned} &\text{for } j = 1, \ldots, f(i) - 1 : && (s_j^i, s_{j+1}^i) \in S, \\ &\text{for } j = 1, \ldots, i : && (v_j^i, s_1^i) \in S, \\ &\text{and} && (s_{f(i)}^i, s_{f(i)}^i) \in S, \end{aligned}$$

which contains a loop at the last element, $s_{f(i)}^i$, and which is linked with all vertices of K_i^f by the first element. All these substructures T_i^f ($i > 0$) share the same relations S and K. An example is shown in Figure 4.4.

Lemma 4.6 *For each computable function $f : \mathbb{N} \to \mathbb{N}$ with $n \leq f(n)$ for all $n \in \mathbb{N}$, DATALOG-NONEMPTINESS($\mathcal{F}^f$) is hard for $DTIME(f)$.*

Proof: We first note that with the following rule, we can use a datalog program with k variables to define a relational successor substructure ranging from 0 to $f(k)$ on $\mathcal{F}^f$. While the EDB S will serve as the successor, the element serving as 0 can be defined with the following rule

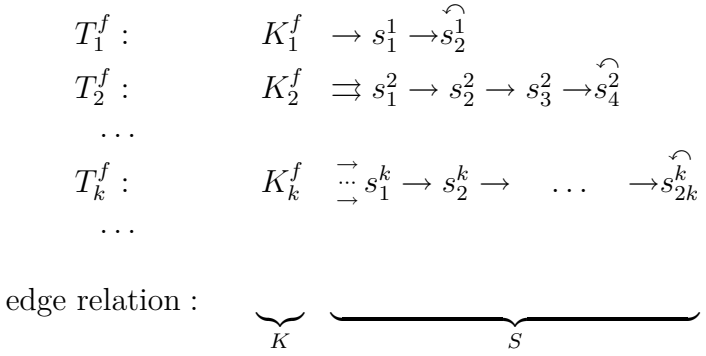


Figure 4.4: The Structure $\mathcal{F}^f$ for $f(n) = 2n$

$$\begin{aligned} \rho: \quad N(x) \leftarrow\ & K(y_1,x), K(y_2,x), \ldots, K(y_k,x), \\ & K(y_1,y_2), K(y_1,y_3), K(y_1,y_4), \ldots, K(y_1,y_k), \\ & K(y_2,y_3), \quad \ldots \quad , K(y_{k-1},y_k). \end{aligned} \tag{4.1}$$

This rule will be satisfied by all K_i with $i \geq k$, so for each datalog program Π containing this rule as only rule with head IDB N, we get

$$N_\infty^\Pi = N_1^\Pi = \left\{ v_j^i \;\middle|\; 1 \leq j \leq i \text{ and } i \geq k \right\} .$$

We can use this relation as "zero" elements for a successor structure (A, S', N), similar to the method used in Corollary 4.2:

$$\begin{aligned} S'(x,y) &\leftarrow S(x,y),\ N(x). \\ S'(x,y) &\leftarrow S(x,y),\ S'(z,x). \end{aligned}$$

This leads to a restriction S' of the relation S to all T_j^f with $j \geq k$. The successor structures in T_j^f each have length at least $f(k)$ and can therefore be used for a Turing machine simulation with at most $f(k)$ steps. Note that more steps cannot be simulated because of the loop $(s_{f(k)}^k, s_{f(k)}^k)$ leading to an ambiguity in the sequence. A simulation continuing for more steps would combine all following steps into one step, allowing different cursor positions and states at the same time.

For the proof of this lemma, assume we are given a one-tape Turing machine M deciding a problem in time $O(f(k))$. By using the linear speedup theorem (see, e.g., [Pap94]), there is an equivalent two-tape Turing machine M' for the problem, running in time $\epsilon f(n) + n + 2$, which (for $\epsilon = 0.5$) is less than $2f(n) \leq f(2n)$ for all $n \geq 2$ by $n \leq f(n)$. The simulation of a two-tape machine works analogously to the simulation of the one-tape machine (see Remark 3.1).

For any input string s of length $n = |s|$, we create the program Π consisting of the simulation of machine M' (which is independent of n), together with the encoding of the input string s (n rules, one for each character of s) and the rule (4.1) for $k = 2n$.

Since the machine M' always halts in less than $f(2n)$ steps, the machine reaches either a state h_{yes} or a state h_{no}, from which no further transition is possible. This automatically restricts our simulation to less than $f(2n)$ steps, which uses the part of all successor structures in S', which does not contain a loop, making the machine simulation a sound simulation. Consequently, $\mathbf{accept}^{\Pi}_{\infty} \neq \emptyset$ if and only if M' (and by equivalence M) accepts. □

4.3.2 Datalog Games

For the upper bound, we need some model theoretic tools to show that for any datalog program the nonemptiness can be decided on a finite part of $\mathcal{F}_r$. In [KV95] for datalog and its extension datalog($\neq$) the connections to finite variable logic and pebble games are studied. To be able to connect datalog to pebble games, it is first shown in Theorem 3.6 of [KV95], that the fixed point of any datalog program Π is definable as a negation-free, inequality-free formula from $L^{m_L+m_R}$, which is the existential negation-free fragment of $L^{m_L+m_R}_{\infty\omega}$. The elements of L^m are obtained from atomic formulas and equalities using infinitary disjunctions, infinitary conjunctions and existential quantification only, using a fixed set $\{x_1, \ldots, x_m\}$ of variables.

This logical definition of a datalog fixed point leads to a test of computability by datalog programs using a pebble game. Using the game, we can show that on two different structures DATALOG-NONEMPTINESS is equivalent. The result needed here is summarized in Remark 4.12.1 of [KV95] and we will state it briefly, in our notation.

Definition 4.7 ***Existential k-Pebble Game (for implications of positive formulae)***

An existential k-pebble game (for implications of positive formulae, short $(\exists, k)$-game) is a pebble game played by two players, Spoiler and Duplicator, played on two structures $\mathcal{A}$ and $\mathcal{B}$, Spoiler on $\mathcal{A}$ and Duplicator on $\mathcal{B}$. Each player has k pebbles, and we denote Spoiler's pebbles by $\{p_1, \ldots, p_k\}$ and Duplicator's pebbles by $\{q_1, \ldots, q_k\}$. A position in the game is a placement of some the pebbles on $\mathcal{A}$ and $\mathcal{B}$, where $\{p_1, \ldots, p_k\}$ may only be placed on $\mathcal{A}$ and $\{q_1, \ldots, q_k\}$ only on $\mathcal{B}$. For $i = 1, \ldots, k$, the element of $\mathcal{A}$ on which pebble p_i is placed will be denoted by a_i, and the element of $\mathcal{B}$ on which q_i is placed by b_i. Initially no pebble is placed.

The game is played in rounds and each round as follows:

Spoiler picks up some pebble p_j. If it is placed on $\mathcal{A}$, Spoiler removes it from $\mathcal{A}$ and Duplicator removes his corresponding pebble q_j from $\mathcal{B}$. If p_j is not placed on $\mathcal{A}$, Spoiler places it on some element a_j of $\mathcal{A}$ and Duplicator responds by placing his pebble q_j on some element b_j of $\mathcal{B}$.

Let $p_{j_1}, \ldots, p_{j_m}$ be the pebbles placed on $\mathcal{A}$ after round i. Spoiler wins after round i if the mapping $h : a_{j_\ell} \mapsto b_{j_\ell}$ (for $\ell = 1, \ldots, m$) is not a homomorphism from the pebbles on $\mathcal{A}$ to the pebbles on $\mathcal{B}$, i.e.

- *if there are j, j' with $a_j = a_{j'}$ and $b_j \neq b_{j'}$, or*
- *if there is a relation R of $\mathcal{A}$ and a tuple $(a_{\ell_1}, \ldots, a_{\ell_{\mathrm{ar}(R)}})$ with:*

$$(a_{\ell_1}, \ldots, a_{\ell_{\mathrm{ar}(R)}}) \in R^{\mathcal{A}} \quad \text{and} \quad (b_{\ell_1}, \ldots, b_{\ell_{\mathrm{ar}(R)}}) \notin R^{\mathcal{B}}$$

Otherwise, the game continues. Duplicator wins the game if he has a winning strategy that allows him to continue playing "forever", i.e., if Spoiler can never win a round of the game.

Lemma 4.8 *[KV95] Let $\mathcal{A}$, $\mathcal{B}$ structures over the same vocabulary. If duplicator wins the $(\exists, k)$-game on $\mathcal{A}$ and $\mathcal{B}$, then for each datalog program Π with $m_L + m_R \leq k$ and each IDB P of Π the following implication holds:*

$$P_\infty^{\Pi,\mathcal{A}} \neq \emptyset \quad \Longrightarrow \quad P_\infty^{\Pi,\mathcal{B}} \neq \emptyset$$

Consequently, for deciding a DATALOG-NONEMPTINESS($\mathcal{F}^f$) instance of a datalog program Π and its IDB P, we only need to consider a finite substructure, as will be shown in the next lemma.

Lemma 4.9 *Let Π be a datalog program on $\mathcal{F}^f$ for some function $f : \mathbb{N} \to \mathbb{N}$ and let P be an IDB of Π. Let $m = m_L + m_R$. By $\mathcal{F}_m^f$ we denote the finite substructure*

$$\mathcal{F}_m^f = \bigcup_{j=1}^{m} T_j^f$$

of $\mathcal{F}^f$. Then

$$P_\infty^{\Pi,\mathcal{F}^f} \neq \emptyset \quad \Longrightarrow \quad P_\infty^{\Pi,\mathcal{F}_m^f} \neq \emptyset$$

Proof: We show the claim with the use of an existential m-pebble game: Spoiler plays on the structure $\mathcal{A} = \mathcal{F}^f$, Duplicator answers on $\mathcal{B} = \mathcal{F}_m^f$. We prove that Duplicator has a winning strategy which shows the claim by Lemma 4.8.

If Spoiler plays pebbles on substructures T_j^f with $j \leq m$ then Duplicator may copy these moves and therefore the mapping from Spoiler's pebbles to Duplicator's pebbles will always be a homomorphism. The interesting moves are those involving a T_j with $j > m$ — which does not occur in $\mathcal{F}_m^f$. We will show that in this case Duplicator can easily play on the substructure T_m of $\mathcal{F}_m^f$ and still has a winning strategy.

Unlike other pebble games involving partial isomorphisms, we only need a partial homomorphism from the elements $\{a_1, \ldots, a_m\}$ covered by Spoiler's pebbles to the elements $\{b_1, \ldots, b_m\}$ covered by Duplicator's pebbles, so for any two pebbled elements a_i, a_j and answers b_i, b_j the following implications have to be satisfied only:

C1:	$a_i = a_j \Rightarrow b_i = b_j$
C2:	$(a_i, a_j) \in K \Rightarrow (b_i, b_j) \in K$
C3:	$(a_i, a_j) \in S \Rightarrow (b_i, b_j) \in S$

These conditions explicitly allow, that two of Duplicator's pebbles are equal, while Spoiler's corresponding pebbles differ, and that a pair of Duplicator's pebbles is in S (or K), and the corresponding pair of Spoiler's pebbles does not satisfy this membership. This allows Duplicator to answer on the same substructure $\mathcal{F}_m^f$ if Spoiler plays on different substructures, e.g. T_i^f and T_j^f for $i, j > m$.

In fact, for $k > m$ there is a homomorphism $h : S_k^f \to S_m^f$ which maps s_1^k to s_1^m and which will be used to determine Duplicator's move (denoted as types **M3** and **M4**) in S_m^f when Spoiler has played on S_k^f, a substructure that does not exist in Duplicators structure F_m^f. To play according to this homomorphism, Duplicator will pebble an element with the same distance to s_1^m as the distance of Spoiler's pebble to s_1^k, where for great distances the last element $s_{f(m)}^m$ with the S-loop is used.

However, there is no homomorphism from T_k^f to T_m^f because of the K_k^f part, only a partial homomorphism on m elements which is used for the moves of type **M1** and **M2**.

In detail, Duplicator responds as follows, where we add the trivial moves as types **M5** and **M6** for completeness:

Type	Spoiler plays	Duplicator plays
M1	p_i on $p_{i'}$ on $v_j \in K_k^f$	q_i on $q_{i'}$ on K_m^f
M2	p_i on free $v_j \in K_k^f$	q_i on arbitrary free $v_{j'} \in K_m^f$
M3	p_i on s_j^k, $j \leq f(m)$	q_i on s_j^m
M4	p_i on s_j^k, $f(m) < j \leq f(k)$	q_i on $s_{f(m)}^m$
M5	p_i on $a \in \mathcal{F}_m^f$	q_i on a
M6	removes p_i	removes q_i

The term "free" in the description of move **M2** refers to the pebbles of the corresponding player: Spoiler plays on an element where none of Spoiler's pebbles is placed, and Duplicator answers also on a free element. Note that move **M2** is possible because Duplicator has only m pebbles and K_m^f consists of m different elements.

One key fact used in this proof is that if for two elements x, y the condition $(x, y) \in K$ or $(x, y) \in S$ is satisfied, then both have to be from the same part of the structure, i.e. for all $\ell \geq 1$ the following equivalence holds:

$$x \in T_\ell^f \quad \Longleftrightarrow \quad y \in T_\ell^f$$

This can be seen from the definition of S and K in which the components T_i^f are disjoint in terms of S and K.

With this observation, it can be shown directly or by an easy induction that the moves above ensure the validity of the following conditions:

1. $a_i \in K_k^f (\subset T_k^f)$ iff $b_i \in K_m^f (\subset T_m^f)$.
2. If $a_i = a_j \in K_k^f$, then $b_i = b_j \in K_m^f$ holds.
3. If $(a_i, a_j) \in K$, then $(b_i, b_j) \in K$.
4. If $a_i = s_\ell^k$ with $\ell \leq f(m)$, then $b_i = s_\ell^m$.
5. If $a_i = s_\ell^k$ with $\ell > f(m)$, then $b_i = s_{f(m)}^m$.

When these conditions are satisfied, **C2** follows directly from condition 3, while **C1** can be derived from conditions 2, 4, and 5, depending on the positions of a_i and a_j. **C3** can be shown by considering the different cases for an edge $(a_i, a_j) \in S$:

$a_i \in K_k^f$**:** By definition of T_k^f, $a_j = s_1^k$. By condition 1, $b_i \in K_m^f$, and by condition 4, $b_j = s_1^m$, yielding $(b_i, b_j) \in S$.

$a_i = s_\ell^k,\ \ell < f(m)$**:** By definition of T_k^f, $a_j = s_{\ell+1}^k$. Condition 4 yields $b_i = s_\ell^m$ and $b_j = s_{\ell+1}^m$ leading to $(b_i, b_j) \in S$.

$a_i = s_\ell^k,\ f(m) \leq \ell < f(k)$**:** By definition of T_k^f, $a_j = s_{\ell+1}^k$. By condition 5, $b_i = b_j = s_{f(m)}^m$, with the loop $(b_i, b_j) \in S$.

$a_i = s_{f(k)}^k$**:** Then T_k forces $a_j = a_i$ and again by condition 5, $b_i = b_j = s_{f(m)}^m$ with the loop $(b_i, b_j) \in S$.

Our analysis shows that the conditions **C1**, **C2**, and **C3** hold after each round, which are a formal description of a homomorphism from the pebbled elements of $\mathcal{A}$ to those of $\mathcal{B}$. This implies, that Duplicator has a winning strategy on the two structures $\mathcal{F}^f$ and $\mathcal{F}_m^f$, and hence the claim follows. □

Lemma 4.10 *Let $f : \mathbb{N} \to \mathbb{N}$ be a function with $f(n)$ computable in time $O(f(n))$ for all $n \in \mathbb{N}$ and $f(n)^{2n} \leq f(n+1)$ for all $n \in \mathbb{N}$. Then DATALOG-NONEMPTINESS($\mathcal{F}^f$) is solvable in deterministic time $O(f(n))$.*

Proof: In the previous lemma we have shown that if DATALOG-NONEMPTINESS for some program Π has a positive answer on $\mathcal{F}^f$, it also has a positive answer on $\mathcal{F}_m^f$, with $m = m_L + m_R$. Obviously, if the answer is negative on $\mathcal{F}^f$ then also on the substructure $\mathcal{F}_m^f$ by the monotonicity of datalog. Hence, it suffices to solve the problem on the substructure $\mathcal{F}_m^f$, which contains

$$\#(\mathcal{F}_m^f) = \sum_{i=1}^{m} (i + f(i)) \leq m^2 + mf(m) \leq 2mf(m)$$

elements.

For computations with the structure $\mathcal{F}^f$ (and also $\mathcal{F}^f_m$ as a substructure) we may represent each element x as a three-tuple $\mathrm{rep}(x) = (i, t, j) \in \mathbb{N} \times \{0, 1\} \times \mathbb{N}$ with the following meaning:

$$x = \begin{cases} k_j \in K^f_i, & \text{if } t = 0 \\ s^i_j \in S^f_i, & \text{if } t = 1 \end{cases}$$

With the natural encoding as binary numbers (without leading zeros) with delimiter # over the alphabet $\{0, 1, \#\}$ each element can be encoded in a string whose length is logarithmic in i and j.

If we pre-calculate the values $f_i := f(i)$ for $i = 1, \ldots, m$ and store them (which can be done in time $O(m \cdot f(m))$ by the properties of f), the running time of checking relation memberships for $\mathcal{F}^f_m$ can be kept low: For each pair (x, y) of elements (with representations $\mathrm{rep}(x) = (i_x, t_x, j_x)$ and $\mathrm{rep}(y) = (i_y, t_y, j_y)$),

- $(x, y) \in K$ if and only if $i_x = i_y$ and $t_x = t_y = 0$,
- $(x, y) \in S$ if $i_x = i_y$, $t_x = 0$, $t_y = 1$ and $j_y = 1$,
- $(x, y) \in S$ if $i_x = i_y$, $t_x = t_y = 1$, and $i_x + 1 = i_y$,
- $(x, y) \in S$ if $i_x = i_y$, $t_x = t_y = 1$, and $i_x = i_y = f_{i_x}$,
- $(x, y) \notin S$ and $(x, y) \notin K$ otherwise.

All these operations need time at most polynomial in the representation length, or logarithmic in i and j.

As we operate on a finite structure, we may instantiate all program rules of Π with all possible values for each variable, which is also known as "grounding". In this process we may omit tuples not occurring in the EDB relations corresponding to the atoms in the rule bodies. Each symbol P with arity $n_P > 0$ is then replaced by a set $\{P_i\}$ of symbols with cardinality at most $\#(\mathcal{F}^f_m)^{m_R}$. Letting $n := |\Pi|$ and recalling that the representation of each element of the structure is $O(\lg f(m))$ in size, the resulting program Π' has the following properties:

- $n' := |\Pi'| \in O(n(mf(m))^{m_R} \lg f(m))$
- For any IDB P of Π:

$$P^\Pi_\infty \neq \emptyset \iff \text{for some } P_i,\ (P_i)^{\Pi'}_\infty \text{ can be derived from the EDB facts}$$

During the grounding process, we may use the precomputed values f_i for the enumeration of tuples to avoid the running time for calculating function values of f.

Since the variables were eliminated by grounding, we can use an algorithm for propositional logic programs running in time polynomial in n', i.e. in $O(n'^\ell)$ for some ℓ, to check whether the IDB symbol P in question admits a derivation in Π' starting from the EDB facts.

This gives us a solution to the problem "$P_\infty^{\Pi} \neq \emptyset$?" running in deterministic time

$$O\left(n'^{\ell}\right) \subseteq O\left(f(m)^{2m}\right) \subseteq O\left(f(n)\right),$$

where the last inclusion uses the fact that $f(m)^{2m} \leq f(m+1)$ and $n > m$. □

We summarize the results of this section in the following theorem:

Theorem 4.11 *Let $f : \mathbb{N} \to \mathbb{N}$ be a function with $f(n)$ computable in time $O(f(n))$ for all $n \in \mathbb{N}$ and $f(n)^{2n} \leq f(n+1)$ for all $n \in \mathbb{N}$.*

Then DATALOG-NONEMPTINESS($\mathcal{F}^f$) is hard for $DTIME(f)$ and is included in $DTIME(f)$.

As one conclusion we transfer this theorem to the context of complexity classes, showing that the dichotomy conjecture (Conjecture 4.4) does not hold. We choose a set of fast growing functions:

Definition 4.12 *For $r, i \in \mathbb{N}$ let $\mathrm{tow}(r,i)$ be the modified tower of twos of height r defined as*

$$\mathrm{tow}(r,i) := 2^{\cdot^{\cdot^{\cdot^{2^i}}}}\Big\}r \tag{4.2}$$

Note that for $r \geq 3$ a straightforward calculation shows the property $\mathrm{tow}(r,i)^{2i} \leq \mathrm{tow}(r,i+1)$ needed for Theorem 4.11.

The complexity classes we use for our hierarchy are the classes with iterated exponential running times, rEXP, also known as exponential hierarchy (see, e.g., [Pap94]). Recall that the complexity class rEXP for $r \in \mathbb{N}$ is defined as

$$r\mathrm{EXP} := \mathrm{DTIME}\left(2^{\cdot^{\cdot^{\cdot^{2^{n^k}}}}}\Big\}r\right) \tag{4.3}$$

for constant values k. The collection of all these classes is known as the class of elementary languages, while the lowest level 1EXP coincides with EXPTIME considered in later chapters.

Corollary 4.13 *For each $r \in \mathbb{N}$ with $r > 2$, let*

$$\mathcal{H}_r := \mathcal{F}^{f_r} \quad \text{for} \quad f_r(n) := 2^{\cdot^{\cdot^{\cdot^{2^n}}}}\Big\}r \ .$$

Then DATALOG-NONEMPTINESS($\mathcal{H}_r$) is hard for $(r-1)$EXP and is contained in rEXP, or shorter

$$(r-1)\mathrm{EXP} \subseteq \mathit{datalog}_{\mathrm{non}-\emptyset}(\mathcal{H}_r) \subseteq r\mathrm{EXP}$$

Due to the exponent k occurring in the definition of the rEXP classes, we unfortunately have to introduce an exponential gap between lower and upper bound. Although we do not have a nice hierarchy of problems which are complete for the corresponding classes, our construction suffices to refute our dichotomy conjecture, since the rEXP classes are a true hierarchy of strictly included classes.

4.4 Summary of this Chapter

In this chapter, we have used the universal Turing machine simulation to show some undecidability results, namely:

1. DATALOG-NONEMPTINESS and DATALOG-TUPLE are undecidable for programs with IDB arity at most two.

The interesting fact is a dichotomy we have exhibited here: As it is well known for monadic programs, i.e. arity one, datalog programs on successor structures are decidable. For arity two we have shown the undecidability, leading to the undecidability of datalog programs with maximal IDB arity at least two, on successor structures.

In the next section, we had a look at some successor-like structures, on which datalog remains undecidable:

2. DATALOG-NONEMPTINESS is undecidable on *infinite trees.*

3. DATALOG-NONEMPTINESS is undecidable on *successor structures which contain some limited set of cycles.*

This result suggests, that even though a positive existential formalism cannot distinguish elements easily, properties of a structure which would usually render a successor structure useless, can be employed if they appear in some regular manner.

In the following section we had a look at a possible connection between undecidability and boundedness of programs and showed by a counter example, that the following direct connection does not exist:

> (*Does not hold.*) If on a structure there exists an unbounded program, then DATALOG-NONEMPTINESS is undecidable on this structure.

In the remaining part of the chapter, we had a look at a conjecture, which might be suggested by the upper bound results of Chapter 5 and the undecidability results:

> (*Does not hold.*) On infinite structures without constants, DATALOG-NONEMPTINESS is either solvable in EXPTIME or it is undecidable.

We refuted this conjecture by creating a set of structures, one for each class of the exponential hierarchy. Unfortunately, we have an exponential gap between the lower and upper bound for the complexity of DATALOG-NONEMPTINESS for each of these structures.

For showing the upper bounds we used the tool of pebble games to reduce the computation to a finite part of the structure.

Open problems

What we started in our discussion about successor like structures, automatically leads to the question:

Question 4.1 *What are the properties of a structure needed, such that it can be used to define an infinite successor structure using datalog?*

Although not needed for breaking a dichotomy, an immediate question to Section 4.3 is:

Question 4.2 *Can the exponential gap between upper and lower bound be narrowed or even be closed?*

Chapter 5

Types and Upper Bounds

Outline of this Chapter

This chapter includes some of the key ideas of this thesis. First, we introduce the concept of special types and show how they allow us to describe datalog computations on (infinite) linear orders. As first application of this concept, we solve DATALOG-NONEMPTINESS and DATALOG-TUPLE for some restricted cases with weakened orders or datalog syntax. Following that, the first interesting application addresses DATALOG-NONEMPTINESS on linear orders. These methods are then refined to show the uniform boundedness of datalog programs on orders and to show the decidability of DATALOG-TUPLE. After dealing with the scope of general linear orders, we use the special case of dense linear orders to derive a tighter bound for DATALOG-TUPLE in this case and to give a detailed example of an algorithm using the type concept to evaluate datalog computations. A section with a direct link with the following chapter will deal with a brief look at the expansion of linear orders by additional constants and how to modify the type concept for this variant. Then we will have a brief look at datalog$^\neg$ in this context. We conclude the chapter with a look at the problem how the representation of the underlying structure influences the complexity and decidability of DATALOG-TUPLE. Before this discussion, we will always assume, that we have a structure oracle as introduced in Section 2.5.

The concept of gap order constraints introduced in [Rev93] is somewhat similar to our distance type concept. The gap order constraints are combined to gap trees to describe datalog computations over the ordered integers and this concept is used to show the decidability of DATALOG-TUPLE($\mathcal{A}$) on ordered integers, using a method which is a simpler version of the one we introduce in Section 5.4.

Some parts of this chapter have been submitted for publication in [GS], mainly from Sections 5.1, 5.3, and 5.6.

5.1 Distance Types

Types are a model theoretic tool that we shall use for dealing with datalog programs on infinite orders. We define an appropriate notion of type and prove a lemma that links them with the evaluation of datalog programs. These types enable us to derive a solving algorithm for DATALOG-NONEMPTINESS($\mathcal{A}$) on linear orders $\mathcal{A}$ which is independent of the representation of $\mathcal{A}$.

Definition 5.1 ***(Distance Atoms and Types)***

1. *A* distance atom *is an expression of the form $x \leq_d y$, $min \leq_d x$, or $x \leq_d max$, where x, y are variables and d is a nonnegative integer. We may write $<_d$ instead of $\leq_d$ for $d > 0$. A* distance type *is a finite set of distance atoms that contains at most one atom $t \leq_d u$ for each pair (t, u) of variables or min, max.*

   We write $\delta(x_1, \ldots, x_k)$ to indicate that the variables of the distance type δ are among $x_1, \ldots, x_k$. The set of all distance types with variables among $x_1, \ldots, x_k$ is denoted by $\Delta(x_1, \ldots, x_k)$.

2. *Let $\mathcal{A} = (A, <)$ be a linear order, $\bar{a} = (a_1, \ldots, a_k) \in A^k$. Let $\delta = \delta(x_1, \ldots, x_k)$ be a distance type. Then $(\mathcal{A}, \bar{a})$* satisfies *δ (we write: $\mathcal{A} \models \delta(\bar{a})$),[1] if*

   - *for all atoms $x_i \leq_d x_j \in \delta$, there are $b_0, \ldots, b_d \in A$ such that $a_i \leq b_0 < b_1 < \ldots < b_d \leq a_j$ (that is, $x_i \leq_d x_j$ is interpreted as $x_i \leq x_j$ and $d(x_i, x_j) \geq d$);* [2]
   - *for all atoms $min \leq_d x_j \in \delta$, there are $b_0, \ldots, b_d \in A$ such that $b_0 < b_1 < \ldots < b_d \leq a_j$;*
   - *for all atoms $x_i \leq_d max \in \delta$, there are $b_0, \ldots, b_d \in A$ such that $a_i \leq b_0 < b_1 < \ldots < b_d$.*

   A distance type δ is satisfiable *if there is a linear order $\mathcal{A}$ and a tuple $\bar{a}$ such that $(\mathcal{A}, \bar{a})$ satisfies δ.*

3. *The* rank *of a distance atom $t \leq_d u$ is d, and the rank of a distance type δ is the maximum of the ranks of all atoms it contains. The set of all distance types in $\Delta(x_1, \ldots, x_k)$ of rank at most d is denoted by $\Delta_d(x_1, \ldots, x_k)$.*

4. *Let $\mathcal{A} = (A, <)$ be a linear order, $\bar{a} = (a_1, \ldots, a_k) \in A^k$, and $d \geq 0$. The* distance-d type of $\bar{a}$ in $\mathcal{A}$, *denoted by $\mathrm{tp}_d(\mathcal{A}, \bar{a})$, is the distance type that contains:*

   - *for $1 \leq i, j \leq k$ such that $a_i \leq a_j$ the distance atom $x_i \leq_c x_j$, where $c = \min\{d, d(a_i, a_j)\}$;*

[1] Another common terminology is to say that a type is *"realized"* instead of "satisfied".

[2] For the definition of the function d, see Section 2.4.

- *for $1 \leq j \leq k$ the distance atom $min \leq_c x_j$, where $c \leq d$ is maximum such that there exists $b_0, \ldots, b_c \in A$ with $b_0 < \ldots < b_c \leq a_j$;*
- *for $1 \leq i \leq k$ the distance atom $x_i \leq_c max$, where $c \leq d$ is maximum such that there exists $b_0, \ldots, b_c \in A$ with $a_i \leq b_0 < \ldots < b_c$.*

5. *A distance type δ is* complete *if there exists a linear order $\mathcal{A}$, a tuple $\bar{a}$ with $\mathcal{A} \models \delta(\bar{a})$, and $d \geq 0$ such that for each pair (a_i, a_j) of entries of $\bar{a}$ satisfying $a_i < a_j$ there is a distance atom $a_i \leq_c a_j$ in δ with $0 < c \leq d$, and for each pair (a_i, a_j) with $a_i = a_j$ there are distance atoms $a_i \leq_0 a_j$ and $a_j \leq_0 a_i$ in δ.*

   The set of all complete distance types with variables among $x_1, \ldots, x_k$ is denoted by $\Gamma(x_1, \ldots, x_k)$, and the set of all types in $\Gamma(x_1, \ldots, x_k)$ of rank at most d is denoted by $\Gamma_d(x_1, \ldots, x_k)$.

Example 5.2 *An example for a distance type from $\Delta(x, y, z)$ is:*

$$\delta = x <_3 y,\ y <_2 z$$

This type δ is satisfied for some elements $a_1, a_2, a_3 \in A$, which we assign to the variables $x := a_1$, $y := a_2$ and $z := a_3$, if there exist $b_1, b_2, b_3 \in A$ with

$$\begin{aligned} a_1 < b_1 < b_2 < a_2 \quad & \text{to satisfy } x <_3 y \\ a_2 < b_3 < a_3 \quad & \text{to satisfy } y <_2 z\ . \end{aligned}$$

The occurring ranks of the atoms in delta show $\delta \in \Delta_3(x, y, z)$. δ is not complete, since there is no distance atom containing x and z and no distance atom containing min *or* max.

Let us point out some subtleties of these definitions that may be confusing. A distance type need not be satisfiable, but a complete distance type must be satisfiable.[3] A distance type may contain atoms $x \leq_c y$ and $y \leq_d x$, but it must not contain atoms $x \leq_c y$ and $x \leq_d y$ for $c \neq d$. If a satisfiable distance type contains the atoms $x \leq_c y$ and $y \leq_d x$, then $c = d = 0$, and the two atoms force x and y to be equal. Even though the "constants" *min* and *max* appear in distance atoms, they are not part of the datalog language, and we do not require linear orders to have a minimum or maximum. The semantics of the atoms $min \leq_d x$, or $x \leq_d max$ is well-defined in all linear orders.

Note that $x \leq_1 y$ is equivalent to $x < y$ and that $x \leq_0 y \wedge y \leq_0 x$ is equivalent to $x = y$. A distance type of rank 1 only contains information about the relative order of the variables and about equalities between the variables, and not about their distances. Hence we call distance types of rank 1 *order types*.

A distance type in n variables contains at most $n^2 + 2n$ atoms. Furthermore, it is easy to see that it can be decided in polynomial time in the number of variables whether a distance type is satisfiable and whether it is complete.

[3] In model theory, it is common to define types as being satisfiable sets of formulas.

Every distance type of rank d can be written as a disjunction of complete types of rank d. That is, for every $\delta \in \Delta_d(x_1, \ldots, x_k)$ there exists a set $\Gamma \subseteq \Gamma_d(x_1, \ldots, x_k)$ such that for all linear orders $\mathcal{A}$ and all tuples $\bar{a} \in A^k$,

$$\mathcal{A} \models \delta(\bar{a}) \iff \text{there exists a } \gamma \in \Gamma \text{ such that } \mathcal{A} \models \gamma(\bar{a}). \tag{5.1}$$

This can be seen by a short inductive argument: If δ is not complete, then we first calculate the transitive closure of the order atoms of δ, i.e. for all variables x_i, x_j, x_ℓ such that δ contains an atom $x_i <_d x_j$ and an atom $x_j <_{d'} x_\ell$, but no atom of the form $x_i <_{d''} x_\ell$, we add the atom $x_i <_1 x_\ell$. After calculating this transitive closure and replacing δ by it, we check if δ is complete. If not, then there are variables x_i and x_j such that δ does not contain a distance atom $x_i <_d x_j$, and we create a set Γ of three types: The first $\delta_1 := \delta \cup \{x_i <_1 x_j\}$, the second $\delta_2 := \delta \cup \{x_j <_1 x_i\}$, the third $\delta_3 := \delta \cup \{x_i = x_j\}$ (modelled by $x_i \leq_0 x_j$ and $x_j \leq_0 x_i$). From the construction, it is clear that the tuples satisfying δ are exactly those satisfying the transitive closure and those satisfying one of the three new types. If the types in Γ are not complete, then they have at least one more distance atom than δ each and we may inductively create sets Γ_1, Γ_2 and Γ_3 of complete distance types of rank d, which are equivalent to δ_1, δ_2 and δ_3, respectively. Then the set Γ is the union $\Gamma_1 \cup \Gamma_2 \cup \Gamma_3$.

A similar idea is used in Lemma 5.12 to modify a datalog program over linear orders in a way, that the order type of each IDB is unique.

Note that if δ is not satisfiable, then Γ is the empty set. Thus we may focus on complete types.

For a complete type $\gamma \in \Gamma(x_1, \ldots, x_k)$ we define the function $D_\gamma : \{x_1, \ldots, x_k\}^2 \cup \{(min, x_i), (x_i, max) \mid 1 \leq i \leq k\} \to \mathbb{N}$ by

$$D_\gamma(t, u) = \begin{cases} d & \text{if } t \leq_d u \in \gamma, \\ 0 & \text{otherwise.} \end{cases}$$

Letting $x_{-1} = \min$ and $x_0 = \max$, the set $\Gamma(x_1, \ldots, x_k)$ is partially ordered by the relation $\preceq$ defined by

$$\gamma \preceq \gamma' \iff D_\gamma(x_i, x_j) \leq D_{\gamma'}(x_i, x_j) \text{ for } -1 \leq i, j \leq k.$$

The following lemma is immediate:

Lemma 5.3 *Let $\mathcal{A} = (A, <)$ be a linear order, $\bar{a} \in A^k$, and $\gamma, \gamma' \in \Gamma(x_1, \ldots, x_k)$ such that $\gamma \preceq \gamma'$. Then*

$$\mathcal{A} \models \gamma'(\bar{a}) \implies \mathcal{A} \models \gamma(\bar{a}).$$

Recall that for each IDB P of a datalog program Π over some fixed structure $\mathcal{A}$, by P_i^Π we denote the interpretation of P after the ith stage of the computation of Π. In the following lemma we will show how to describe the stages by finite sets of distance types, but first we will have a look at a simple example.

Example 5.4 *We have a look at the following two rule program* Π *over linear orders* $\mathcal{A} = (A, <)$*, defining IDBs* P *and* Q*:*

$$P(x, y) \leftarrow x < z_1,\ z_1 < z_2,\ z_2 < y.$$
$$Q(x, y, z) \leftarrow P(x, y),\ y < w,\ w < z.$$

Applying the first rule to the empty IDB relations at the beginning, the resulting relation P_1^Π *contains all tuples satisfying the distance type* $x <_3 y$*, since there have to be three distance atoms satisfied between the elements assigned to* x *and* y*.*

Applying the second rule to this stage 1, this distance type is copied to the type describing the tuples in Q_2^Π *and on* y *and* z *the type* $y <_2 z$ *is imposed, leading to the following type describing* Q_2^Π*:*

$$\delta = x <_3 y,\ y <_2 z$$

For programs using recursion and more rules leading to some form of disjunction, a single distance type is not enough to describe a relation, but sets of types are needed.

Lemma 5.5 *Let* $\mathcal{A} = (A, <)$ *be an infinite linear order and* Π *a datalog program over* $\mathcal{A}$*.*

Then for each k*-ary IDB* P *of* Π *and each* $i \geq 0$ *there is a finite set* $\Theta(P, i) \subseteq \Gamma(x_1, \ldots, x_k)$ *such that for all* $\bar{a} \in A^k$ *it holds that*

$$\bar{a} \in P_i^\Pi \iff \text{there is a } \theta \in \Theta(P, i) \text{ such that } \mathcal{A} \models \theta(\bar{a}).$$

Furthermore, the rank of all types in $\Theta(P, i)$ *is bounded by* $(m_R)^i$*, where* m_R *denotes the maximal number of variables in a rule of* Π *as usual.*

Proof: Recall that we compute the IDB relations applying exactly on rule in each stage i.

We prove this claim by induction on i. The induction base for $i = 0$ is obvious: We let $\Theta(P, 0) = \emptyset$ for all IDBs P.

For the induction step ($i > 0$), we apply a rule ρ. Depending on the form of ρ, we distinguish between two cases. In the first case, the rule ρ is of the form

$$\rho:\ P(x_1, \ldots, x_k) \leftarrow E(x_1, \ldots, x_k, y_1, \ldots, y_m).$$

E is a conjunction of EDB atoms of the form $z_1 < z_2$. To derive an equivalent distance type $\gamma \in \Gamma(x_1, \ldots, x_k, y_1, \ldots, y_m)$, we represent E as a directed graph G: The vertices of G are the variables $\{x_1, \ldots, x_k, y_1, \ldots, y_m\}$, and there is an edge (z_1, z_2) in G whenever there is an EDB atom $z_1 < z_2$ in E. If there is a directed cycle in G, then obviously E is not satisfiable and we let $\Theta(P, i + 1) = \Theta(P, i)$.

Otherwise, if for some variables w_1 and w_2 as nodes of G, there is a directed path from w_1 to w_2, then there is a longest such path, say $w_1 \to v_1 \to v_2 \to \cdots \to v_d \to w_2$, and each satisfying assignment of E has to satisfy the following inequalities:

$$w_1 < v_1 < v_2 < \cdots < v_d < w_2$$

It follows that each satisfying assignment has to satisfy $w_1 < w_2$ and $d(w_1, w_2) \geq d$, or equivalently $w_1 <_d w_2$.

If we collect all the distance-atoms derived from G (where we use $x <_1 y$ instead of $x < y$), their conjunction $\delta' \in \Delta(x_1, \ldots, x_k, y_1, \ldots, y_m)$ is realized by all satisfying assignments of E. For each variable x_j we add a distance atom $min \leq_d x_j$, where d is the length of the longest path in G ending in x_j; we proceed analogously for $x_j \leq_d max$. Next, we omit all distance atoms containing variables in $\{y_1, \ldots, y_m\}$ from δ' and obtain a distance type $\delta \in \Delta(x_1, \ldots, x_k)$. Then a tuple $\bar{a} \in A^k$ satisfies δ if and only if there is a tuple $\bar{b} \in A^m$ such that $\bar{a}\bar{b}$ satisfies δ'. Now we choose a set $\Gamma \subseteq \Gamma(x_1, \ldots, x_k)$ such that δ is equivalent to the disjunction over Γ, that is, (5.1) holds. We let $\Theta(P, i+1) = \Theta(P, i) \cup \Gamma$. Since there are no cycles in G, the number of variables on the right hand side and hence m_R is a bound on the rank.

In the induction step, we consider rules with IDB atoms in the body. EDB atoms, which may also be present, can be treated similar to IDB atoms in the following way: Let $E(x, y) = x < y$ be an EDB atom (which is the only form of EDB atoms present). Then the distance type of all tuples satisfying this EDB is $x <_1 y$. Thus, we simply let $\Theta(E, i) = \{(x <_1 y)\}$ for all $i \geq 0$, making it unnecessary to distinguish between EDB and IDB atoms in the body, why we will only consider IDB atoms in the following part of the proof.

Suppose ρ is of the form

$$\rho : P(x_1, \ldots, x_k) \leftarrow I(x_1, \ldots, x_k, y_1, \ldots, y_m),$$

where I consists of IDB atoms. Let $\alpha_1, \ldots, \alpha_\ell$ be the atoms in $I(x_1, \ldots, x_k, y_1, \ldots, y_m)$ and let P_j be the IDB of α_j, $j = 1, \ldots, \ell$. By induction hypothesis we have a $\Theta(P_j, i)$ for each α_j.

For each combination of types in $\Theta(P_1, i), \ldots, \Theta(P_\ell, i)$ we define a new set Γ of complete types and add it to $\Theta(P, i+1)$. So let $\gamma_1 \in \Theta(P_1, i), \ldots, \gamma_\ell \in \Theta(P_\ell, i)$.

We create a weighted digraph G representing the conjunction of the γ_j. The vertex set of G is $V = \{x_1, \ldots, x_k, y_1, \ldots, y_m, min, max\}$. The edges are defined by the distance atoms as follows: For each $(t, u) \in V^2 \setminus \{(min, max)\}$ such there is a distance atom $t <_c u$ in a γ_j for some $c \geq 0$ and $j \in \{1, \ldots, \ell\}$, we add an edge (t, u). As all the γ_j are complete types for a subset of variables, for all $(t, u) \in V^2 \setminus \{(min, max)\}$ and $j \leq \ell$ there is a well defined integer $d_j = D_{\gamma_j}(t, u)$. As the weight of edge (t, u) we let $d = \max\{d_j \mid 1 \leq j \leq \ell\}$.

If there is a directed cycle of positive weight in G, then the combination $\gamma_1, \ldots, \gamma_\ell$ of types is not satisfiable, and we let $\Gamma = \emptyset$.

Otherwise we create a distance type $\delta' \in \Delta(x_1, \ldots, x_k, y_1, \ldots, y_m)$. For every pair z_1, z_2 of variables such that there is a directed path

$$z_1 \xrightarrow{d_1} v_1 \xrightarrow{d_2} v_2 \xrightarrow{d_3} \cdots v_l \xrightarrow{d_r} z_2$$

in G (where we choose the path of maximal length $\sum_{j=1}^{r} d_j$ between z_1 and z_2), we add a distance atom $u_1 \leq_c u_2$, where $c = \sum_{j=1}^{r} d_j$. Similarly, we add atoms $min \leq_e z$ and $z \leq_f max$ for the longest path ending at z, and starting at z, respectively. We let $\delta \in \Delta(x_1, \ldots, x_k)$ be the restriction of δ' to the variables $x_1, \ldots, x_k$ and $\Gamma \subseteq \Gamma(x_1, \ldots, x_k)$ a set of complete types such that δ is equivalent to the disjunction over Γ.

For all pairs of variables (x_i, x_j), the satisfying assignments are those satisfying all distance atoms on paths from x_i to x_j, which is equivalent to satisfying the distance atom $x_i <_d x_j$, with d being the length of the longest path from x_i to x_j. That for a satisfying assignment to the head variables $x_1, \ldots, x_k$, a satisfying assignment to the body variables $y_1, \ldots, y_m$ can be found, is ensured by considering all paths between vertices corresponding to head variables.

The proof is concluded with the observation that the maximal rank of a newly introduced distance-atom $z_1 <_{d'} z_2$ is bounded by (applying the induction hypothesis)

$$d' = \sum_{j=1}^{l+1} d_j \leq \sum_{j=1}^{m_R} m_R^i \leq m_R^{i+1} \ .$$

□

This Lemma also gives us a bound on the maximal rank d needed in the program evaluation, but unfortunately this bound depends on the number of fixed point stages and does not lead to a time bound on the computation. It is not obvious that there should be a bound on d which depends only on the program Π and makes it possible to solve DATALOG-NONEMPTINESS using only finitely many fixed point stages. The following example shows that the ranks of the types can increase during a computation in a way that can get quite complicated:

Example 5.6 *Consider the following program consisting of rules ρ_1, ρ_2 and ρ_3, letting $\bar{x} = (x_1, \ldots, x_5)$. We use the abbreviation $x_i <_2 x_j$ for $x_i < y, y < x_j$ omitting some body variable y in the definition of Π which does not appear elsewhere in the rules.*

$$\begin{aligned}
\rho_1 : P\bar{x} \ &\leftarrow x_1 <_2 x_2,\ x_2 <_2 x_3,\ x_4 <_2 x_5.\\
\rho_2 : P\bar{x} \ &\leftarrow x_1 < x_2,\ x_4 < z_2,\ z_3 < y_4,\ y_5 < x_5,\\
&\qquad P(x_2, x_3, z_1, z_2, z_3),\ P(y_1, y_2, y_3, y_4, y_5).\\
\rho_3 : P\bar{x} \ &\leftarrow P(x_1, x_2, x_3, z_1, z_2), P(y_1, x_4, x_5, y_2, y_3).
\end{aligned}$$

The rule ρ_1 is an initialization rule which initializes all distance atoms considered in this example to $<_2$.

The rule ρ_2 introduces $x_1 <_1 x_2$, reuses existing types by copying and sums up some existing atoms from possibly different existing types.

This rule uses two recursive occurrences of the IDB P, which in our description of the application of this rule by types leads to the use of two (possibly different) types from the type set describing earlier stages of P_∞^Π. We denote the ranks of the distance atoms from these two types occurring in our computation, using $\bar{a} = (a_1, \ldots, a_5)$ for tuples from such stages, by:

Ranks in distance types of the earlier stages of P_∞^Π		
first occurrence of P	$a_1 <_{c_1} a_2$	$a_4 <_{c_2} a_5$
second occurrence of P	$a_1 <_{c_1'} a_2$	$a_4 <_{c_2'} a_5$

The rule application of ρ_2 using these distance atoms will then impose the following type on the body variables, where the distance atoms are given in the order of the variable appearance in the rule, omitting the atoms containing the variables z_1, y_1, y_2 and y_3, not part of the result:

$$x_1 <_1 x_2,\ x_4 < z_2,\ z_3 <_1 y_4,\ y_5 <_1 x_5,\ x_2 <_{c_1} x_3,\ z_2 <_{c_2} z_3,\ y_4 <_{c_2'} y_5$$

To combine these types by eliminating non-head variables, we rearrange these atoms:

$$x_1 <_1 x_2,\ x_2 <_{c_1} x_3,\ x_4 <_1 z_2,\ z_2 <_{c_2} z_3,\ z_3 <_1 y_4,\ y_4 <_{c_2'} y_5,\ y_5 <_1 x_5$$

After the elimination of non-head variables, the following type is added to the type set of P:

$$x_1 <_1 x_2,\ x_2 <_{c_1} x_3,\ x_4 <_{c_2+c_2'+3} x_5$$

Rule ρ_3 copies some distance atoms for x_1, x_2, x_3 and transfers some $x_2 <_c x_3$ to $x_4 <_c x_5$ in the result. We conclude the example with the shortest program run leading to a fixed point, described by the ranks of the types $x_1 <_{d_1} x_2$, $x_2 <_{d_2} x_3$ and $x_4 <_{d_3} x_5$. We assume that always the smallest ranks are chosen. Longer runs could lead to even bigger intermediate results, but will have the same final result.

step	*rule*	d_1	d_2	d_3	*remarks*
1	ρ_1	*2*	*2*	*2*	
2	ρ_2	*1*	*2*	*7*	
3	ρ_2	*1*	*1*	*12*	*using tuples in line 2 and 1*
4	ρ_3	*1*	*1*	*1*	*using tuple in line 3 twice*

5.2 Simple Cases

Before proving some interesting and also unexpected time bounds for DATALOG-NONEMPTINESS and DATALOG-TUPLE on linear orders, showing how to deal with cases like the one from the previous Example 5.6, we apply the type concept to some simple cases, showing that no computation power is gained compared to the much simpler propositional horn clauses, which have PTIME-complete complexity (see [DEGV97]).

5.2.1 Non-Strict Orders

The first simplification is to allow the order $\mathcal{A} = (A, <)$ to be non-strict, i.e. there is an element $a \in A$ satisfying $a <^{\mathcal{A}} a$. In this case the lower bound arguments from section 3.2 do not work, since we cannot create two disjoint sets of tuples solely by using the order relation. In general, only the PTIME-hard lower bound remains, which is tight for DATALOG-NONEMPTINESS($\mathcal{A}$). For DATALOG-TUPLE($\mathcal{A}$), we only have the general upper bound proved in Section 5.4, since depending on the tuple entries, we may not take advantage of any of the elements a in the reflexive part of $<^{\mathcal{A}}$.

Of course all arguments of this section also hold for a reflexive order, i.e. $\mathcal{A} = (A, \leq)$ with $a \leq^{\mathcal{A}} a$ for all $a \in A$. In this case, DATALOG-TUPLE($\mathcal{A}$) is similar to the case of a dense order and can be solved in EXPTIME as shown in Section 5.5.

Lemma 5.7 *Over a non-strict linear order $\mathcal{A} = (A, \leq)$ DATALOG-NONEMPTINESS($\mathcal{A}$) can be decided in PTIME.*

Proof: Since the order is not a strict order, each set of EDB relations can be satisfied by a constant assignment of a to all variables, where a is an element satisfying $a <^{\mathcal{A}} a$.

Hence for deciding DATALOG-NONEMPTINESS($\mathcal{A}$) in this case, we only have to check whether P can be derived from EDB atoms. This problem is known from the case of propositional logic programming and can be decided in PTIME. □

5.2.2 Monadic Datalog

Datalog programs having only null-ary and unary IDB symbols are called **monadic datalog programs**. The complexity of the datalog related problems depends on the structure of our order. If we have an order containing a minimal or a maximal element, some simple form of counting is possible. Without these designated elements, monadic datalog programs on orders are not more powerful than propositional logic programs:

Lemma 5.8 *On dense orders $\mathcal{A} = (A, <)$ without endpoints, DATALOG-NONEMPTINESS($\mathcal{A}$) and DATALOG-TUPLE($\mathcal{A}$) restricted to monadic programs are both in PTIME.*

Proof: On dense orders, distance types collapse to order types, and hence by Lemma 5.5 for each IDB symbol P, all elements in P_∞^Π have the same order type. Since P is unary, it follows that either $P_\infty^\Pi = A$ or $P_\infty^\Pi = \emptyset$. Thus, for solving DATALOG-TUPLE($\mathcal{A}$) we may simply solve DATALOG-NONEMPTINESS($\mathcal{A}$).

For this problem an easier version of the algorithm in Lemma 5.14 can be used: An IDB can be made nonempty by applying a rule whose body IDBs are all nonempty and which has consistent body EDBs. Hence each rule only needs to be applied at

most once, and cycling through all rules n_R times solves the problem, altogether in polynomial time (since the consistency check for EDBs can be carried out by employing a digraph-acyclicity test, which is in the complexity class NL). □

Obviously, this result does not depend on the representation of the infinite linear order, not yielding any representation issues, even for the tuple problem.

Now we consider monadic datalog programs on structures $\mathcal{N} = (\mathbb{N}, <)$ with a discrete, strict, linear and infinite order with minimal element denoted by the constant min. We explicitly mention the constant min here to distinguish between ordered structures with and without minimal element. For talking about the relations calculated by the datalog programs we will make use of the constant symbols $\{0, 1, 2, \dots\}$, where $0 = \min$ is the minimal element, 1 its successor and so on. We may also denote the successor of a by $a+1$, and in general the i-th successor of an element a by $a+i$. By ∞ we denote an additional element which is greater than all elements in A or incomparable to all elements in A.

Lemma 5.9 *Let Π be a monadic datalog program over an infinite structure $\mathcal{N} = (\mathbb{N}, <)$ with a discrete, strict, linear order with minimum. Then for each IDB symbol P and the IDB relation P_∞^Π calculated by Π for P there exists $w_P \in \mathbb{N} \cup \{+\infty\}$ with $P_\infty^\Pi = \{x \in \mathbb{N} \mid x \geq w_P\}$.*

Moreover, with the number of rules of Π denoted by n_R and the maximal number of variables per rule in Π denoted by m_R, we have that for all IDB symbols P of Π either $w_P = \infty$ or $w_P \leq n_I \cdot m_R$ holds.

These w_P can all be calculated in time polynomial in n_I and m_R.

Proof: By Lemma 5.5, we know how the types describing the IDB relations look like. In this monadic case, only one variable x and the constant min occur in the types, leading to distance types of the form $\min <_d x$, which can be written as $\{x \in \mathbb{N} \mid x \geq d\}$.

We first apply all rules in an order, such that each rule is applied exactly once and only to an empty head IDB. This order can be calculated by having a look a the dependencies of the rules on the body IDBs and calculating some form of topological sorting of the rules.

It remains to show that $w_P \leq n_I \cdot m_R$ holds for all $w_P < \infty$ and this can be shown by a simple induction argument, similar to the one used in proof of Lemma 5.5. For the induction base, a rule is applied with a body consisting of EDB atoms only. The number of variables in such a rule is bounded from above by m_R, and these body variables can be used to define a distance from min to the head variable x of at most m_R.

For the induction step assume, that after stage i, a rule ρ with head $R(x)$ is applied to an empty IDB R_i^Π and that each IDB P occurring in the rule body of the rule to be applied has a description $P_i^\Pi = \{x \in \mathbb{N} \mid x \geq w_P\}$ with finite w_P. By induction hypothesis, $w_P < i \cdot m_R$ for all these IDBs. Using some of these IDBs as a starting point for a chain of EDB atoms, the distance w'_R of the head variable x to

min can be one of these w_P, increased by the number of body variables, leading to an upper bound of $w'_R \leq \max\{w_P\} + m_R \leq im_R + m_R = (i+1)m_R$.

Further rule applications may decrease the values w_P by at least one per rule application. So we cycle through all rules $n_I^2 \cdot m_R$ times ($n_I \cdot m_R$ times per IDB) and can be sure, that no change will happen afterwards. □

Corollary 5.10 *For monadic programs over an infinite structure $\mathcal{N} = (\mathbb{N}, <)$ with a discrete, strict, linear order with minimum both DATALOG-NONEMPTINESS($\mathcal{N}$) and DATALOG-TUPLE($\mathcal{N}$) are complete for PTIME.*

Proof: While the solution of DATALOG-NONEMPTINESS is just a direct application of the previous Lemma 5.9, for DATALOG-TUPLE representation issues have to be considered. After calculating the bound w_P for all IDBs P, we have to check if for the (unary) input tuple a, $a \geq w_P$ is satisfied. As usual, we employ our structure oracle here, which answers us the question $0 <_{w_P} a$ in constant time. Writing the tuple a and the element 0 on the oracle input tape, takes linear time, which does not influence the asymptotic polynomial running time. □

5.3 DATALOG-NONEMPTINESS on Orders

After considering some simple cases, we return to total linear orders and unrestricted datalog programs. Using the formal description of the IDB relations by distance types and discrete order types we will show an upper bound for DATALOG-NONEMPTINESS. But before, we transform the program into some normal form which integrates the possible order types into the program by creating disjoint copies of each IDB, each having a different order type and hence leading to disjoint relations.

Definition 5.11 *A datalog program over linear orders $\mathcal{A}$ is* type disjoint *if for every k-ary IDB P there is a complete order type $\gamma_P \in \Gamma_1(x_1, \ldots, x_k)$ such that for all linear orders $\mathcal{A} = (A, <)$ and all tuples $\bar{a} \in P_\infty^\Pi$ it holds that $\mathrm{tp}_1(\mathcal{A}, \bar{a}) = \gamma_P$.*

The order type *of an IDB P in a type disjoint program Π is the order type γ_P.*

Lemma 5.12 *For every datalog program P over linear orders there is a type disjoint datalog program Π' over linear orders with the following properties:*

1. *For every IDB P of Π there are IDBs $P_1, \ldots, P_{n_P}$ of Π' of pairwise distinct order types, such that for every linear order $\mathcal{A}$,*

$$P_\infty^\Pi = \bigcup_{j=1}^{n_P} (P_j)_\infty^{\Pi'}.$$

2. *$n'_I \leq n_I \cdot 3^{m_L^2}$, $n'_R \leq 3^{m_R^2 \cdot (m'_I+1)} \cdot n_R$, $m'_R = m_R$ and $m'_L = m_L$, $m_I = m'_I$, where n'_I, m'_R, m'_L, m'_I are the parameters of Π'.*

Furthermore, the program Π' *can be computed from* Π *in exponential time.*

Proof: Of each IDB P of arity r, we create $n_P = 3^{r^2}$ distinct copies $P_0, \ldots, P_{n_P-1}$, each having a different order type. For $i \in \{0, \ldots, n_P - 1\}$, let $(i_0, \ldots, i_{r^2-1})$ be the ternary representation of the number i, $i = \sum_{j=0}^{r^2-1} i_j \cdot 3^j$ with $0 \leq i_j < 3$ for all $j = 1, \ldots, n_P - 1$. Then we link with each new IDB P_i a distance-1-type γ_{P_i} which consists of the following distance atoms:

$$\begin{aligned} x_{j_1} &= x_{j_2}, \quad \text{if } i_{j_1+j_2\cdot r} = 0 \\ x_{j_1} &<_1 x_{j_2}, \quad \text{if } i_{j_1+j_2\cdot r} = 1 \\ x_{j_2} &<_1 x_{j_1}, \quad \text{if } i_{j_1+j_2\cdot r} = 2 \end{aligned}$$

So each combination of distance atoms for all pairs of variables will be present in some γ_{P_i}. After computing these distance types, we transform the program in two stages. First, we change the head IDBs to the new IDB set consisting of the distinct copies created as above for each IDB of Π: Each rule ρ with head atom $P\bar{x}$ is replaced by copies $\rho'_0, \ldots, \rho'_{n_P-1}$ with head $P_j\bar{x}$, and the body copied from ρ and extended by EDBs $x_{j_1} < x_{j_2}$ for each $(x_{j_1} <_1 x_{j_2}) \in \gamma_{P_j}$. In case of $(x_{j_1} = x_{j_2}) \in \gamma_{P_j}$ we replace all occurrences of x_{j_2} by x_{j_1} afterwards which simulates the equality relation not available as EDB or IDB relation.

Each of the rules $\rho_0, \ldots, \rho_{n_P-1}$ is then itself replaced by copies which instead of the body IDBs from Π, use the IDBs of Π':

Of each rule ρ_r, say, $P_l\bar{x} \leftarrow Q_1\bar{y}_1, \ldots, Q_m\bar{y}_m, E(\bar{x}, \bar{y}_1, \ldots, \bar{y}_m)$, with E being a sequence of EDBs, each IDB Q_j of Π has been converted to a set of IDBs $\{Q_{j\ell}\}$. From these sets we generate all possible combinations $(Q_{1\ell_1}, \ldots, Q_{m\ell_m})$ and create from each combination a rule of Π':

$$P_l\bar{x} \leftarrow Q_{1\ell_1}, \ldots, Q_{m\ell_m}, \; E(\bar{x}, \bar{y}_1, \ldots, \bar{y}_m) \;,$$

where the sequence E of EDB atoms is left untouched.

After that, we directly eliminate a rule with an inconsistent order type. This can be done by viewing rule as graph with the variables being the nodes and the order atoms being directed edges. A check for a directed cycle, which can be carried out in time polynomial in the rule length, shows if the order type is inconsistent.

Each tuple added to a stage in the evaluation of Π introduced by some rule ρ of Π has a complete distance 1 type, so there will be one of the copies of ρ which can be applied to add this tuple. Conversely, the newly created rules of Π' may only add tuples for which a rule in Π exists adding this tuple.

So each IDB of arity r is converted to not more than 3^{r^2} copies, each with a different order type, and hence $n'_I \leq n_I \cdot 3^{m_L^2}$. For each rule, we need all combinations of copies of the newly created IDBs, adding up to at most $n'_R \leq \left(3^{m_L^2}\right)^{(m_I+1)} \cdot n_R = 3^{m_L^2 \cdot (m_I+1)} \cdot n_R$. □

To demonstrate the idea behind Lemma 5.12, we give a short example showing how a program is converted into type disjoint form.

Example 5.13 *We consider a program* Π *over a linear order* $\mathcal{A} = (A, <)$ *whose rules are given in the first column of the following table. The second column lists the type disjoint copies of each rule (together forming program* Π'*) with the corresponding order type in the third column.*

Rules of Π	*Type disjoint rules of* Π'	*Order type*
$P(x,y,z) \leftarrow x<z,\ y<z.$	$P_1(x,y,z) \leftarrow x<z,\ y<z,\ x<y$	$x<y<z$
	$P_2(x,y,z) \leftarrow x<z,\ y<z,\ y<x.$	$y<x<z$
	$P_3(x,x,z) \leftarrow x<z,\ x<z.$	$x(=y)<z$
$Q(x,y) \leftarrow P(x,y,w).$	$Q_1(x,y) \leftarrow P_1(x,y,w).$	$x<y<w$
	$Q_2(x,y) \leftarrow P_2(x,y,w).$	$y<x<w$
	$Q_3(x,x) \leftarrow P_3(x,x,w).$	$x(=y)<w$

In the first rule, the order type is not fixed for x *and* y*, leaving the relative order of these two variables open. The type disjoint versions of the rule will catch all occurring cases.*

In this example P *will be replaced by* P_1, P_2, P_3 *and* Q *by* Q_1, Q_2, Q_3*.*

For type disjoint datalog programs, DATALOG-NONEMPTINESS can be solved in a simple fashion, essentially disregarding any recursion in rules. In the following lemma we construct an execution sequence s that will suffice to decide DATALOG-NONEMPTINESS.

Lemma 5.14 *Let* Π *be a type disjoint datalog program over an infinite linear order* $\mathcal{A} = (A, <)$*. Then there exist an* $i_s \leq n_I$ *and a sequence* $s = (\rho_0, \rho_1, \ldots, \rho_{i_s-1})$ *of rules, such that after applying* ρ_i *to stage* i *for* $i = 0, \ldots, i_s - 1$*, the emptiness is determined, i.e. for all IDBs* P *it holds that*

$$P_{i_s}^{\Pi,\mathcal{A}} = \emptyset \quad \Rightarrow \quad P_{\infty}^{\Pi,\mathcal{A}} = \emptyset \ . \tag{5.2}$$

This sequence s *can be computed in time polynomial in the program length.*

Proof: We create the sequence s by cycling through the rules n_I times, adding those rules to s which change an empty IDB to nonempty. Formally:

$s = (\rho_0, \rho_1, \ldots, \rho_{i_s-1})$ such that there exist IDBs $P_1, \ldots, P_{i_s}$ with $(P_i)_i^{\Pi} = \emptyset$, and after applying ρ_i, $(P_i)_{i+1}^{\Pi} \neq \emptyset$ for $i = 0, \ldots, i_s - 1$.

We continue this process until no more rules can be applied to make an empty IDB nonempty, but this can happen at most n_I times, immediately leading to the time bound for the computation. Note that nonempty IDBs are never modified by s.

The crucial observation is that in a type disjoint program it only depends on the nonemptiness of the IDBs in the body if a rule adds new tuples to the previously empty head IDB, and not on the actual content of the body IDBs. Each rule ρ to be instantiated (in some stage i) has some EDB atoms and some IDB atoms corresponding to nonempty IDB atoms in the body. Since the program is type disjoint, there is a complete order type for each IDB which all tuples in this IDB relation have to

satisfy. If we combine the order types of all IDBs together with the EDB atoms, the resulting type τ has to be satisfied by each instantiation of ρ. If τ is consistent (i.e. satisfiable) can be checked by a test running in time polynomial in the rule length (e.g. by creating a directed graph with an edge for each order atom which is then checked for acyclicity).

Since all IDBs $P_1, \ldots, P_k$ of ρ have been made nonempty by a rule application before, by Lemma 5.5 there is a distance type δ_{P_j} such that all tuples in $(P_j)_i^\Pi$ satisfy δ_{P_j} and since $\mathcal{A}$ has an infinite universe, for each distance type δ with $\delta_{P_j} \preceq \delta$, all tuples satisfying δ are in $(P_j)_i^\Pi$. Because of this, we can create a *satisfiable* distance type δ_ρ from the EDBs and the distance types $\delta_{P_1}, \ldots, \delta_{P_k}$ as in Lemma 5.5, and all assignments to the variables of ρ satisfying δ_ρ will be instantiations of ρ. Hence the head IDB can be made nonempty.

We now show property (5.2) by contradiction:

Let $U = \left\{ R \;\middle|\; R_{i_s}^\Pi = \emptyset \wedge R_\infty^\Pi \neq \emptyset \right\}$ be the set of IDBs changing to nonempty after s and assume $U \neq \emptyset$. Then for each $R \in U$ there exist an $i_R \in \mathbb{N}$ and a rule ρ_R with:

$$R_{i_R}^\Pi = \emptyset, \text{ and applying } \rho_R \text{ to } R_{i_R}^\Pi : \; R_{i_R+1}^\Pi \neq \emptyset \;.$$

Let $P \in U$ be the IDB with $i_P = \min \{ i_R \mid R \in U \}$. By the definition of U and by the choice of i_P, all $Q \in U \setminus \{P\}$ have to satisfy $Q_{i_P}^\Pi = \emptyset$. Since a rule can be applied if and only if all body IDBs are nonempty, the rule ρ_R cannot depend on them and can be applied in stage i_P leading to a sequence of rule applications making more IDBs nonempty, a contradiction to the construction of s. □

Corollary 5.15 ***(Order Invariance of DATALOG-NONEMPTINESS)***
Let $\mathcal{A} = (A, <^\mathcal{A})$ and $\mathcal{B} = (B, <^\mathcal{B})$ be two infinite linear orders. Then for each datalog program Π and each IDB P of Π,

$$P_\infty^{\Pi,\mathcal{A}} = \emptyset \quad \Leftrightarrow \quad P_\infty^{\Pi,\mathcal{B}} = \emptyset$$

Proof: By Lemma 5.12 we may create a type disjoint version of Π, not depending on all properties of the underlying structure, but only the order axioms. Thus, this type disjoint version will be the same for the two different linear orders $\mathcal{A}$ and $\mathcal{B}$ and we may assume Π to be in type disjoint form.

As shown in Lemma 5.14, we create an initialization sequence s for Π on $\mathcal{A}$. Then $P_\infty^{\Pi,\mathcal{A}} = \emptyset$ holds iff $P_{i_s}^{\Pi,\mathcal{A}} = \emptyset$ after the application of s.

We inductively show how to apply the initialization sequence to $\mathcal{B}$. In stage 0, a rule ρ_0 is applied, which is possible over $\mathcal{A}$, because only EDB atoms occur in the body of ρ_0 and all variables have a satisfiable order type. But then also in $\mathcal{B}$ there will be elements with that order type, making it possible to apply ρ_0 over $\mathcal{B}$.

For the induction step, assume that $P_j^{\Pi,\mathcal{A}} \neq \emptyset$ implies $P_j^{\Pi,\mathcal{B}} \neq \emptyset$ for all $j \leq i$. In stage i the rule ρ_i is applied over $\mathcal{A}$ to make some IDB R nonempty, $R_{i+1}^{\Pi,\mathcal{A}} \neq \emptyset$, so

all body IDBs occurring in this rule must be nonempty and there are elements in A satisfying the order type γ of the variables of rule ρ_i (combined from the order types of the IDBs and EDBs of this rule). But since $\mathcal{A}$ and $\mathcal{B}$ are both orders, there will also be elements in B with this order type γ, so also over $\mathcal{B}$ the rule ρ_i can be applied and $R_{i+1}^{\Pi,\mathcal{B}} \neq \emptyset$ follows.

Since relations becoming nonempty at some stage stay nonempty, we get:

$$P_\infty^{\Pi,\mathcal{A}} \neq \emptyset \quad \Rightarrow \quad P_{i_s}^{\Pi,\mathcal{A}} \neq \emptyset \quad \Rightarrow \quad P_{i_s}^{\Pi,\mathcal{B}} \neq \emptyset \quad \Rightarrow \quad P_\infty^{\Pi,\mathcal{B}} \neq \emptyset$$

The same argument holds for an initialization sequence $s_\mathcal{B}$ on $\mathcal{B}$, leading to

$$P_\infty^{\Pi,\mathcal{B}} \neq \emptyset \quad \Rightarrow \quad P_\infty^{\Pi,\mathcal{A}} \neq \emptyset$$

□

Theorem 5.16 *DATALOG-NONEMPTINESS($\mathcal{A}$) over linear orders $\mathcal{A} = (A, <)$ is EXPTIME-complete.*

Proof: The proof is a combination of several earlier results. A datalog program Π can by Lemma 5.12 be converted to a type disjoint program Π'. For this kind of program Lemma 5.14 gives us a method to check which IDB relations of Π' will be empty after an evaluation of Π'. Since Π' is type disjoint, each IDB relation of the original program Π will occur here as a collection of IDBs of Π', which can easily be determined. Thus, the question "$P_\infty^\Pi = \emptyset$?" can be answered by checking the type sets of all corresponding IDBs of Π'. Beside the time for this check and the time for the conversion of the programs, the time for determining the empty IDB relations of Π' is part of the running time. Using Lemma 5.12 and Lemma 5.14 the time of this step is at most exponential in the program length, altogether in EXPTIME and with the earlier shown EXPTIME-hardness the claim follows. □

5.4 Boundedness and DATALOG-TUPLE on Orders

A datalog program Π is *bounded on a structure* $\mathcal{A}$ if there is a computation of Π on $\mathcal{A}$ that reaches a fixed point after finitely many stages. Of course, this concept of boundedness is nontrivial only on infinite structures. The main result of this section is that datalog programs are bounded on linear orders. Actually, we prove a stronger result giving a uniform bound on the number of evaluation steps that is computable from the size of the program and does not depend on the structure. This discussion is part of the boundedness context for finite structures introduced in the introduction.

Definition 5.17 *Let Π be a datalog program over a structure $\mathcal{A}$.*

1. *A* computation sequence *for* Π *is a sequence* s *of rules of* Π *to compute all IDB relations, i.e. a sequence of rules satisfying the following conditions:*
   - *If* s *is finite, then after applying* s*, no further rule application adds a tuple to the IDB relations.*
   - *If* s *is infinite, then each rule of* Π *will occur infinitely often.*
2. *The* closure ordinal *of* Π *on* $\mathcal{A}$*, denoted by* $\mathrm{cl}(\Pi, \mathcal{A})$*, is the length of the shortest computation sequence for* Π *on* $\mathcal{A}$ *(*$\mathrm{cl}(\Pi, \mathcal{A}) = \infty$ *if all computation sequences are of infinite length).*
3. Π *is* bounded on $\mathcal{A}$ *if* $\mathrm{cl}(\Pi, \mathcal{A}) < \infty$*.*

Now let C *be a class of structures such that* Π *is a program over* C*.*

3. *The* uniform closure ordinal *of* Π *on* C*, denoted by* $\mathrm{ucl}(\Pi, C)$*, is the maximum of the closure ordinals* $\mathrm{cl}(\Pi, \mathcal{A})$ *for* $\mathcal{A} \in C$ *if this maximum exists, and* ∞ *otherwise.*
4. Π *is* uniformly bounded on C *if* $\mathrm{ucl}(\Pi, C) < \infty$*.*

For each program, one computation sequence is the sequence simply cycling through the rules in some fixed order, but there may be shorter computation sequences. The condition on infinite computation sequences that each rule has to occur infinitely often, ensures that for unbounded programs the simultaneous fixed point of all IDB relations is computed correctly and all IDB relations are considered in the computation.

Note that if Π is uniformly bounded on C, then it is bounded on all $\mathcal{A} \in C$, but that the converse does not necessarily hold.

Theorem 5.18 *Let* Π *be a datalog program over the class* LO *of linear orders. Then* Π *is uniformly bounded on* LO*. More precisely, there is a computable function* $b : \mathbb{N} \mapsto \mathbb{N}$ *such that for* $n = |\Pi|$ *it holds that*

$$\mathrm{ucl}(\Pi, LO) \leq b(n).$$

Our proof of Theorem 5.18 is based on a simplification of the distance type concept which we will discuss before the presentation of the main proof. The proof presented here is an extension of the proof of Theorem 5.16, first transforming the program Π in question to a type disjoint version Π' by Lemma 5.12 and then creating the initialization sequence s as in Lemma 5.14. After this process, we may eliminate all then empty IDB relations.

Each remaining IDB P may only contain tuples of one complete order type ϑ_P, and if we denote the arity of P by n_P, this complete order type ϑ_P consists of exactly $n_P(n_P - 1)/2$ atoms: Denoting the tuple entries of tuples in P by $(x_1, \ldots, x_{n_P})$, for any pair (x_i, x_j) with $i < j$, the order type ϑ_P contains either $x_i < x_j$, $x_i = x_j$, or

$x_j < x_i$. Each distance type γ in the type set $\Theta(P,i)$ can then be represented as this order type ϑ_P and additionally a vector of ranks, each corresponding one of the inequality atoms in the order type and transforming the order atom into a distance atom.

In the following, we use this representation of each distance type γ by a *rank vector* $\bar{d}^\gamma = (d_1^\gamma, \ldots, d_{k_P}^\gamma)$ which together with the order type of P leads to a well defined description of $\gamma \in \Theta(P,i)$. The entries of the rank vector can be created by applying the function D_γ (introduced for Lemma 5.3) to each pair of variables occurring in an atom of the order type ϑ_P in some fixed order. The sets $\Theta(P,i)$ can then be represented as finite sets of rank vectors

After applying the sequence s and after eliminating empty IDBs, for each IDB P, the set $\Theta(P,i_s)$ is described by exactly one such vector, since $|\Theta(P,i_s)| = 1$ after the initialization sequence which adds at most one type to the type set of each IDB. By Lemma 5.3, all tuples realizing a type γ' with $\gamma \preceq \gamma'$, also realize the weaker type γ. Hence increasing the size of an IDB relation P by adding new tuples (which realize a newly added type γ') is only possible if in all present types $\gamma \in \Theta(P,i)$ some atom rank of γ is greater than the corresponding rank in γ', i.e. $\gamma \not\preceq \gamma'$.

In terms of rank vectors, a type γ defines a set $\mathcal{H}_\gamma$ containing the vectors of all types γ' that are at least as restrictive as γ, i.e. $\gamma \preceq \gamma'$:

$$\mathcal{H}_\gamma = \{(x_1, \ldots, x_{k_P}) \mid x_\ell \geq d_\ell^\gamma \text{ for } \ell = 1, \ldots, k_P\}$$

Speaking of rank vectors, $\gamma \preceq \gamma'$ if the rank vector $\bar{d}^\gamma$ is *dominated* by the rank vector $\bar{d}^{\gamma'}$, i.e. for all $i = 1, \ldots, k_P$, $d_i^\gamma \leq d_i^{\gamma'}$. Then $\mathcal{H}_\gamma$ is the set of all types with a rank vector dominating the rank vector of γ.

Then creating a sequence of new types added to $\Theta(P,i)$ is equivalent to the search for a non-dominating sequence of rank vectors, where we call a (finite or infinite) sequence $x_1, x_2, \ldots$ *non-dominating* if for all i and j with $i < j$, x_j does not dominate x_i.

Figure 5.1 shows a graphical representation for $k_P = 2$. Figure 5.1 (c) shows a case where a new vector is added containing a coordinate greater than the maximum of all existing entries. But this growth can only occur in a limited manner, as we will show. Before, we introduce some notations.

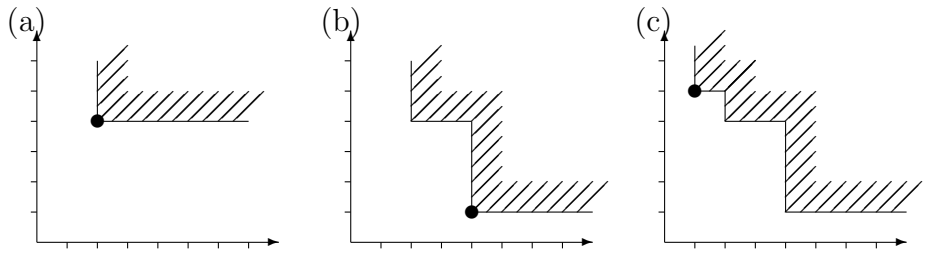
Definition 5.19

Let $k \in \mathbb{N}$ and $\bar{x} = (x_1, \ldots, x_k) \in \mathbb{N}^k$. Then $||\bar{x}||_\infty := \max\{x_1, \ldots, x_k\}$. For $S \subset \mathbb{N}^k$, let $||S||_\infty := \max_{\bar{x} \in S} ||\bar{x}||_\infty$. Let $s_1, \ldots, s_\ell$ be finite sequences, each sequence consisting of tuples of some arity, and let $C = (s_1, \ldots, s_\ell)$ be a tuple of these sequences. Then $||C||_\infty := \max_{i=1}^{\ell} ||s_i||_\infty$, where the sequences are considered as sets.

To model the rank vectors occurring in the stages of the IDB relations, we introduce a corresponding sequence concept:

Definition 5.20 ***(c-Bounded Run)***

*Let $t \in \mathbb{N}$, let $k_1, \ldots, k_t \in \mathbb{N}$ and for $i = 1, \ldots, t$ let $\bar{x}_i \in \mathbb{N}^{k_i}$. Let $c \in \mathbb{N}$. Then X is a c-*bounded run of $(\bar{x}_1, \ldots, \bar{x}_t)$*, if*



Example of the description of an IDB relation with rank vectors of length 2 (x and y coordinate).
Figure (a) shows a description with one rank vector, automatically including all types with rank vectors in the hatched area. Figure (b) shows the situation after a second rank vector was added, automatically including more types. In Figure (c) a vector with greater entry than the entries before is added.

Figure 5.1: Geometric Representation of a Type Set

- *$s_1^0, \ldots, s_t^0$ are sequences of tuples, where for each i, s_i^0 consists of the tuple $\bar{x}_i$ only.*
- *The stage X_0 of X is the tuple $X_0 = (s_1^0, \ldots, s_t^0)$.*
- *Inductively, the j-th stage $X_j = (s_1^j, \ldots, s_t^j)$ of X is created from the stage $X_{j-1} = (s_1^{j-1}, \ldots, s_t^{j-1})$ by choosing an $\ell \in \{1, \ldots, t\}$, a $\mu_j \in \mathbb{N}$, and $\{\bar{x}_1, \ldots, \bar{x}_{\mu_j}\} \subset \mathbb{N}^{k_\ell}$ such that*
  - $\mu_j \leq (||X_0||_\infty \cdot c^{j-1})^{c^2}$
  - *for $n \neq \ell$: $s_n^j = s_n^{j-1}$*
  - $s_\ell^j = s_\ell^{j-1} \circ (\bar{x}_1, \ldots, \bar{x}_{\mu_j})$ *($\circ$ meaning sequence concatenation)*
  - *s_ℓ^j is non-dominating*
  - $||x||_\infty \leq c \cdot ||X_{j-1}||_\infty$

The condition on μ_j ensures that the sequence added in each stage is finite and bounded from above by some function of $||X_0||_\infty$, c and j, which will be needed for the computation of a uniform bound.

The connection between the setting of datalog programs on orders and the c-bounded runs is given by the following lemma:

Lemma 5.21 *Let t be the number of nonempty IDB relations of the type disjoint program Π' after the initialization sequence s of length i_s from Lemma 5.14. Then for each nonempty IDB relation P, the set $\Theta(P, i_s)$ contains exactly one rank vector. Let $\bar{d}_1, \ldots, \bar{d}_t$ be these rank vectors. Let $m = \max\{m'_R, m'_I, m'_L\}$.*

1. *For all $j = 1, \ldots, t$: $||\bar{d}_i||_\infty \leq (m'_R)^{i_s} \leq (m'_R)^{n'_R n'_I}$.*

2. *For each computation of* Π' *continuing the initialization sequence, the rank vectors added during this computation form an* m*-bounded run* X *of* $(\bar{d}_1, \ldots, \bar{d}_t)$.

Proof:

1. $||\bar{d}_i||_\infty \leq (m'_R)^{i_s}$ follows from Lemma 5.5 and $(m'_R)^{i_s} \leq (m'_R)^{n'_R n'_I}$ follows from Lemma 5.14.

2. This is a similar argument as in Lemma 5.5, where a directed graph is created to derive a distance type.

   At stage i, a rule ρ with some head IDB P is applied, which leads to a type γ that has a rank vector not dominated by a rank vector of an existing type in $\Theta(P, i)$. The rule has some head variables $\{x_1, \ldots, x_k\}$ and body variables $\{x_1, \ldots, x_k, y_1, \ldots, y_{k'}\}$. Then for creating the new type γ, for each IDB occurrence in ρ a distance type from $\Gamma(x_1, \ldots, x_k, y_1, \ldots, y_{k'})$ is used; EDBs as order atoms can be converted to the equivalent distance-1-atoms. We call this set of types Δ and use it to build a weighted directed graph G_Δ:

   The vertices of G_Δ are the variables $\{x_1, \ldots, x_k, y_1, \ldots, y_{k'}\}$ and the edges are created in the following way: For each pair (z_1, z_2) of variables, such that there exists a distance atom $z_1 \leq_d z_2$ in Δ for some $d \geq 0$, there is an edge (z_1, z_2) in G_Δ. The edge (z_1, z_2) has as weight the maximal value d over all distance atoms $z_1 \leq_d z_2$ in Δ.

   From this graph the type γ is created: For each pair (x_i, x_j) of variables, such that G_Δ contains a directed path from x_i to x_j, γ contains the distance atom $x_i <_d x_j$, where d is the length (sum of edge weights) of the longest path from x_i to x_j in G_Δ. As shown in Lemma 5.5, this atom $x_i <_d x_j$ can be satisfied if and only if all distance atoms that correspond to edges on paths from x_i to x_j can be satisfied, which corresponds to finding a satisfying assignment for all variables of rule ρ.

   The rank vector of γ satisfies the property of an element added in a stage of a c-bounded run: By construction all edges in G_Δ have weight at most $||X_i||_\infty$. The longest paths used for creating the distance atoms in γ, use at most all of the vertices $\{x_1, \ldots, x_k, y_1, \ldots, y_{k'}\}$ (where $k + k' \leq m'_R$), hence the longest paths have length at most $m'_R \cdot ||X_i||_\infty \leq m \cdot ||X_i||_\infty$ and the claim follows.

   For applying rule ρ in stage i, for each body IDB, we have finitely many choices for a type (by Lemma 5.5). For each combination, we derive a type γ with some rank vector $\bar{d}_\gamma$. Then all these rank vectors form a finite set Ξ. Since all these vectors are introduced in the same stage, independently of each other, we may order them in a finite sequence ξ in an arbitrary order. We then make this sequence non-dominating by removing the dominated entries and name the created sequence ζ. The removed rank vectors correspond to some distance types which are weaker than some existing types, and hence we do not change

the solution set by removing these types. Thus, ζ satisfies the properties of $\bar{x}$ in the definition of c-bounded runs.

It only remains to show the bound on μ_{j+1}. By the last property of Definition 5.20, we know that in stage j of the m-bounded run it holds that $||X_j||_\infty \leq m^j||X_0||_\infty$ and hence in each sequence s_j^i there are no more than $||X_j||_\infty^m \leq (m^j||X_0||_\infty)^{m_L}$ elements (since the maximal IDB arity m_L is a bound on the arities of all sequences $s_1^i, \ldots, s_t^i$). Each sequence contains the rank vector of the types in $\Theta(P, i_s + j)$ for some IDB P, so these type sets have no more elements than this bound. There are at most m'_I IDB occurrences in each rule ρ, hence no more than

$$\mu_{j+1} \leq ((m^j||X_0||_\infty)^{m'_L})^{m'_I} \leq (m^j||X_0||_\infty)^{m^2}$$

different types can be created with one rule application to stage j of the run, as claimed.

□

To show a computable uniform bound on c-bounded runs, we need two well known lemmas which we state here without proof:

Lemma 5.22 ***König's Tree Lemma*** *(see, e.g., [Hod97; Die00])*
Let T be an infinite *rooted directed tree with finite branching (i.e. each vertex has a finite number of children). Then T contains an infinite path starting at the root node.*

Lemma 5.23 ***Dickson's Lemma*** *(see, e.g., [HMS06; Dic13])*
All non-dominating sequences of tuples of natural numbers are finite.

These finiteness (or infiniteness) properties allow us to compute a bound on the number of stages of c-bounded runs:

Lemma 5.24 *There is a nondecreasing computable function $f : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ with the following property:*
For all $m \in \mathbb{N}$, $c \in \mathbb{N}$, $t \in \mathbb{N}$, $r \in \mathbb{N}$, $k_1, \ldots, k_t \in \{1, \ldots, r\}$ and $\bar{x}_i \in \mathbb{N}^{k_i}$ with $||\bar{x}_i||_\infty \leq m$, each c-bounded run of $(\bar{x}_1, \ldots, \bar{x}_t)$ has at most $f(m, c, t, r)$ stages.

Proof: First, we have a look at an arbitrary choice of m, c, t, r, $k_1, \ldots, k_t$ and $\bar{x}_i$ (for $i = 1, \ldots, t$).

We create a labeled tree T containing all c-bounded runs of these values: The root node is labeled with X_0. Inductively, for each node labeled with a stage X_i, we create a child node for each stage X_{i+1} created from X_i and label it with the corresponding stage.

To create a stage X_{i+1} from X_i, we may choose each of the t sequences to extend it. Each sequence s_j^i has an arity k_j and by the last condition of a c-bounded run, the

element added to this sequence may only have coordinates that are at most $c||X_i||_\infty$. Because of this and since the length of μ_{i+1} of the extension of the sequence in stage $i+1$ is bounded from above by $(||X_0||_\infty \cdot c^{j-1})^{c^2}$, there are only finitely many choices for a finite extension of a sequence and hence finitely many children for each node in this tree T (each for a different extension of some sequence).

A path in T (starting at the root node) corresponds to one c-bounded run. We now show that each path is finite: Assume, we have an infinite path p in T. This path p is labeled with the stages of a c-bounded run X. In each stage one of the sequences of X is extended by finitely many elements and since there are only t sequences there has to be one sequence that is extended in infinitely many stages. Each extension of this sequence is non-dominating, so we get an infinite non-dominating sequence. But by Dickson's Lemma each non-dominating sequence of tuples of natural numbers is finite, a contradiction. Hence all paths in T are finite.

Hence T has finite branching (only finitely many children to each node) and no infinite path. By König's Lemma T must be finite.

The height of T is the greatest number of stages that can occur in a c-bounded run of $\bar{x}_1, \ldots, \bar{x}_k$. Since T is finite, we can compute the whole tree and determine its height.

We discuss how to compute the value $f(m, c, t, r)$ for given values m, c, t and r:

For each fixed choice $k_1, \ldots, k_t$ of arities, the entries of all choices of the corresponding tuples $\bar{x}_1, \ldots, \bar{x}_t$ are bounded by m and thus for each tuple there are only finitely many choices. By computing the height of the tree (by creating the tree) to each choice of tuples one after the other and determining the maximum $h(k_1, \ldots, k_t)$, we have computed a bound on the number of stages for the c-bounded runs with sequence arities $k_1, \ldots, k_t$.

The parameter t determines the number of sequences in the runs considered and the parameter r limits the arities of these sequences. The maximum of the values $h(k_1, \ldots, k_t)$ over all possible r^t sequence arity tuples $(k_1, \ldots, k_t)$ is then the maximal number of stages in a c-bounded run with t sequences and sequence arities at most r and it can be computed by computing $h(k_1, \ldots, k_t)$ for all finitely many choices.

This maximum satisfies the properties of $f(m, c, t, r)$, and that f is nondecreasing is immediate: Increasing some parameter, all runs remain valid, but also longer runs may appear. □

This function will directly lead to the function b of Theorem 5.18:

Proof: [of Theorem 5.18] The program Π over a linear ordering $\mathcal{A} = (A, <)$ is first converted to an equivalent type disjoint version Π' as in Lemma 5.12, which also gives the bounds $n'_I \leq n_I \cdot 3^{m_L^2}$, $n'_R \leq 3^{m_L^2 \cdot (m_I+1)} \cdot n_R$, $m'_R = m_R$ and $m'_L = m_L$ for the parameters of the new program Π'. Then the initialization sequence s as in Lemma 5.14 is determined, resulting in the first $i_s \leq n'_I \leq n_I \cdot 3^{m_L^2}$ stages.

While the empty relations of Π' can be neglected, each nonempty relation P of Π' has a type description $\Theta(P, i)$ with one rank vector each, and by Lemma 5.21 these rank vectors satisfy the properties of an m_R-bounded run X. By Lemma 5.24, X has at most $f((m'_R)^{n'_R n'_I}, m'_R, n'_I, m'_L)$ stages.

We let $b(n) := f(n^{3^{n^4}}, n, n \cdot 3^{n^2}, n)$ and by the above bounds on the parameters and since each program parameter is bounded from above by the program length n, $f((m'_R)^{n'_R n'_I}, m'_R, n'_I, m'_L) \leq f(n^{3^{n^4}}, n, n \cdot 3^{n^2}, n) \leq b(n)$ and the claim follows.

Since $\mathcal{A}$ was chosen as arbitrary linear order, this bound also holds for $\mathrm{ucl}(\Pi, \mathrm{LO})$. □

Let us remark that, combined with Lemma 5.5 and its proof, Theorem 5.18 implies that DATALOG-TUPLE is decidable for computable linear orders. Recall that DATALOG-TUPLE($\mathcal{A}$) asks if a given tuple of elements of $\mathcal{A}$ is in the fixed-point of a given IDB of a given program Π.

Corollary 5.25 *On linear orders $\mathcal{A} = (A, <)$ DATALOG-TUPLE($\mathcal{A}$) is decidable.*

Proof: By Theorem 5.18, we can compute the distance type set describing the relation $P_\infty^{\Pi,\mathcal{A}}$ for each IDB P of Π. To solve DATALOG-TUPLE($\mathcal{A}$) for the instance $(\Pi, P, \bar{a})$, all types describing one of the type disjoint copies of P have to be considered.

To check if $\bar{a} \in P_\infty^{\Pi,\mathcal{A}}$ holds, we have to select the disjoint copy P_j of P whose order type is satisfied by $\bar{a}$. The then following check, whether a distance type describing P_j is satisfied by $\bar{a}$, means that for each distance type, we have to iterate over its distance atoms and ask a oracle query with the corresponding tuple entries.

Following Theorem 5.18 we only need a finite number of oracle queries, each with a finite representation of its input (the tuple entries and the rank of the distance atom). Hence the whole procedure runs in finite time. □

5.5 Datalog on Dense Orders

On dense linear orders $\mathcal{A} = (A, <)$ without endpoints, we can match the singly exponential lower bound by using some form of quantifier elimination: The main difference in this case is, that for each two elements $a, b \in A$, there are infinitely many different elements $c_1, c_2, \ldots$ with $a <^{\mathcal{A}} c_1 <^{\mathcal{A}} c_2 <^{\mathcal{A}} \cdots <^{\mathcal{A}} b$. This makes it easy to satisfy a positive existential formula, in contrast to orders not having the denseness property, where there may be only constantly many or even no elements between two selected elements a and b. Let *DLO* denote the class of dense linear orders without endpoints.

Theorem 5.26 *Let Π be a datalog program over the class of linear orders and $n = |\Pi|$. Then*

$$\mathrm{ucl}(\Pi, DLO) \leq 3^{n^2}.$$

Proof: Observe that on dense linear orders, distance types collapse to order types, because for all a, b and for all $d \geq 1$ we have $a < b \iff a \leq_d b$. So the only types to consider are distance-1-types (including equality atoms, when we consider complete distance-1-types) and as the type disjoint version of a program as introduced in Lemma 5.12 contains all complete distance-1-types, it contains all types of interest

for this case. After evaluating the initialization sequence as computed in Lemma 5.14, all complete distance-1-types describing the IDB relations are computed and hence no rule can be applied after that to add new types.

Since there are at most 3^{n^2} different distance-1-types for a program of length n, the claim follows. □

As an example of the representation of order types in algorithms, we give a more detailed version of this theorem together with a detailed proof.

Lemma 5.27 *Let* Π *be a datalog program over a dense linear order without endpoints* $\mathcal{A} = (A, <)$. *There is a deterministic algorithm which calculates the sets of order types for all IDB relations of* Π *with* $n = |\Pi|$ *in time*

$$O(n_R m_I m_R^2 3^{m_L^2 m_I + m_L^2}) \subseteq O(3^{2n^3}) \tag{5.3}$$

Proof: We will give an algorithm which is closely related to the proof of Lemma 5.5 and whose correctness will mainly follow directly from this proof. Then we give some details and running time bounds for the algorithm.

To be able to calculate distance-1-types, it is necessary to specify how these can be stored. A distance-1-type τ of k variables is a collection of atoms $v_1 < v_2$ or $v_1 = v_2$ with variables in $\{x_1, \ldots, x_k\}$. To encode this we set the representation $\mathrm{rep}(\tau)$ of τ to be the 0-1-string of length $2 \cdot k^2$ having

$$\mathrm{rep}(\tau)[j + ki + 1] = \begin{cases} 1, & \text{if } (x_{i+1} < x_{j+1}) \in \tau \\ 0, & \text{otherwise,} \end{cases}$$

$$\mathrm{rep}(\tau)[k^2 + j + ki + 1] = \begin{cases} 1, & \text{if } (x_{i+1} = x_{j+1}) \in \tau \\ 0, & \text{otherwise,} \end{cases}$$

where $0 \leq i, j < k$. The representation is combined of two parts: The first k^2 bits denote the presence of inequality atoms, while the rest denotes equality atoms. A complete type (which is satisfiable by definition) contains for each pair (x_i, x_j) of variables with $i \neq j$ exactly one of the atoms $x_i < x_j$, $x_i = x_j$, $x_j < x_i$ and therefore a set of distance-1-types of the variables $\{x_1, \ldots, x_k\}$ may contain at most 3^{k^2-k} elements.

Since these types can be ordered by ordering their representation strings, an ordered set structure can be used which supports the iteration over its elements and adding elements (avoiding duplicate entries). As implementation, e.g. a balanced search tree can be used, supporting the iteration to the next element in amortized constant time and the add operation in logarithmic time.

The following algorithm calculates for each IDB P in Π the such a set T_P of order types describing P_Π. After the run of the algorithm, we have $T_P = \Theta(P, \infty)$ for each IDB P. By n_P we denote the arity of the IDB P.

Input: datalog program Π
Output: T_P for each IDB P in Π
1: **for all** IDBs P in Π **do**
2: $T_P := \emptyset$ % Initialization
3: $M := \prod_{\text{IDB } P \text{ in } \Pi} 3^{n_P^2}$
4: **for** $i := 1$ **to** M **do** % or less, if fixed point is reached
5: **for all** Rules ρ of Π **do**
6: $P :=$ head IDB of ρ
7: $\tau_E :=$ type of EDBs of ρ % over all body variables
8: Let $I_1, \ldots, I_s$ be the IDB symbols in the body of ρ
9: **for all** $(\tau_1, \ldots, \tau_s) \in T_{I_1} \times \cdots \times T_{I_s}$ **do** % with s iterators
10: $\tau_{\text{new}} := \tau_E$
11: **for** $j := 1$ **to** s **do** % Create union of types
12: $\tau :=$ Expansion of τ_j to variables of τ_E
13: $\tau_{\text{new}} := \tau_{\text{new}} + \tau$ % + is bitwise or
14: **if** τ_{new} can be realized **then**
15: $\tau' :=$ Projection of τ_{new} to variables of P
16: $T_P.add(\tau')$ % New type found

This algorithm calculates the order type describing the fixed point containing the IDB relations of the program Π. The outer loop is used for calculating all stages of this fixed point. Since there can be at most $3^{n_P^2}$ different types for an IDB P, at most this many steps are needed for each IDB, summing up to M as defined in line 3 for the total number of stages until the fixed point is reached. Of course, the number of executions of this loop could be limited further if a check for changes was implemented. But since that does change the worst case running time, we omit this refinement here.

Lines 6 to 16 of the algorithm simulate the application of rule ρ and set the describing sets of order types accordingly. In line 7 the type equivalent to all EDBs occurring in ρ is calculated and represented as 0-1-string. Then all possible combinations of types of IDB relations are used to generate a new type which is up to expansions and renamings the union of the types. This is done by representing the type of an IDB relation as 0-1-string, expand it to the body variables and then setting all positions in the new type τ_{new} to 1 which are set in τ.

The resulting 0-1-string representing τ_{new} is then used to add a projection of this type to the variables of P to the set T_P. This is only done if τ_{new} can be realized, since inconsistency informations could be lost by the projection.

The realizability test can easily be done interpreting the type as graph $G = (V, E)$ with the variables as vertices, where equality atoms will be used to combine different variables into the same vertex, e.g. an atom $v_1 = v_2$ will cause the vertex of v_1 to be used for v_2. The inequality atoms $v_1 < v_2$ are translated to directed edges (v_1, v_2). Then the type can be realized, if and only if the graph is acyclic. This graph is then also used to model the equality atoms in τ_{new} correctly, ensuring that for each equality atom $v_1 = v_2$ the variables v_1 and v_2 satisfy the same inequalities.

To conclude the proof we will now derive an upper bound for the running time of this algorithm.

In line 7 the representation of τ_E is generated for which all EDBs are read and transferred into the vector τ_E and then the transitive closure is built, using $O(m_R^3)$ steps.

In lines 12 and 13, the bits from the representation of τ_j are copied to the representation of τ and then added to the representation of τ_{new}, which can be done in $O(m_R^2)$ time. The realizability test can be done in time $O(m_R^2)$, while line 15 takes $O(m_R^2)$ time and line 16 can be carried out in time $O(\log(4^{m_L^2})) \subseteq O(m_L^2)$, since each of the T_P may only contain less than $4^{m_L^2}$ different types. So for lines 10 to 16, we have $O(m_I \cdot m_R^2)$ steps.

Lines 9 to 16 are executed at most $(3^{m_L^2})^{m_I} = 3^{m_L^2 \cdot m_I}$ times, resulting in a total running time of $O(m_I m_R^2 3^{m_L^2 \cdot m_I})$ steps for this loop. For the whole algorithm the running time adds up to

$$O(3^{m_L^2} \cdot n_R \cdot m_I m_R^2 3^{m_L^2}) \subseteq O(n_R m_I m_R^2 3^{m_L^2 m_I + m_L^2}) \subseteq O(3^{2n^3}) \ ,$$

since each parameter is bounded from above by the program length n. □

Corollary 5.28 *Over a dense linear order $\mathcal{A}$ without endpoints and without constants DATALOG-TUPLE($\mathcal{A}$) can be decided in EXPTIME.*

Proof: For given program Π, tuple a and IDB P, we want to know if $a \in P_\Pi$ holds.

By using the algorithm of 5.27, we calculate the set T_P. For each type $\tau \in T_P$ we can then derive the representation $\text{rep}(\tau)$ and check if a realizes τ. For this test, we employ our structure oracle for order types. There at most exponentially many types, each containing at most a quadratic number of order atoms, yielding an altogether exponential number of order queries. For each query, the input are two of the tuple entries, which can clearly be written to the oracle tape in linear time. Thus, the running time of the whole algorithm is at most singly exponential. □

Together with corollary 3.3 we have shown that these two problems are EXPTIME-complete on (strict) dense linear orders.

5.6 Datalog on Orders with Constants

We may also consider datalog programs over an ordering $\mathcal{A} = (A, <, c_1, \ldots, c_r)$ incorporating finitely many constants $c_1, \ldots, c_r$, each of them being interpreted as a fixed element of A, and allow the datalog programs to make use of these constant symbols.

To solve DATALOG-NONEMPTINESS or DATALOG-TUPLE for such a program Π with constants, we transform the program to a constant free version Π' by essentially replacing each constant c_i by a fresh variable and adding rule parts to transfer the values of all constants to all rules. Using this technique, we show:

Theorem 5.29

DATALOG-TUPLE($\mathcal{A}$) on linear orders $\mathcal{A} = (A, <, c_1, \ldots, c_r)$ with finitely many constants (which may be used by the datalog programs) is decidable.

Proof: We first transform the program Π over $\mathcal{A} = (A, <, c_1, \ldots, c_r)$ to a program Π' over the structure $\mathcal{A}' = (A, <)$ increasing the arity of each IDB symbol by r, such that for each IDB P of Π with arity s and its corresponding IDB P' of Π', and each $\bar{a} = (a_1, \ldots, a_s) \in A^s$ the following holds:

$$\bar{a} \in P_\infty^{\Pi,\mathcal{A}} \qquad \Longleftrightarrow \qquad \bar{c}\bar{a} = (c_1^{\mathcal{A}}, \ldots, c_r^{\mathcal{A}}, a_1, \ldots, a_s) \in (P')_\infty^{\Pi',\mathcal{A}'} \tag{5.4}$$

This is established by replacing all occurrences of the constant c_i by a fresh variable C_i, for $i = 1, \ldots, r$. To ensure that all rule applications share the same values for the constants, we augment each IDB P of Π by the additional variables $\bar{C} = (C_1, \ldots, C_r)$ and replace all occurrences of $P(\bar{x})$ by $P'(\bar{C}, \bar{x})$ — in the rule bodies and the rule heads. This forces the values of the variables $C_1, \ldots, C_r$ to be identical in the body and head of each rule and hence in the whole program. For example, if $\phi(c_1, \ldots, c_r, x_1, \ldots, x_m, y_1, \ldots, y_n)$ is the formula appearing in the body of the rule

$$P(x_1, \ldots, x_m) \;\leftarrow\; \phi(c_1, \ldots, c_r, x_1, \ldots, x_m, y_1, \ldots, y_n).$$

we translate this rule to:

$$P'(C_1, \ldots, C_r, x_1, \ldots, x_m) \;\leftarrow\; \phi(C_1, \ldots, C_r, x_1, \ldots, x_m, y_1, \ldots, y_n).$$

It is now clear, that the original program Π and the modified version Π' satisfy the condition (5.4).

This transformation can be carried out in logarithmic space, since the number r of constants does only depend on the structure $\mathcal{A}$, not the input $(\Pi, P, \bar{a})$ of DATALOG-TUPLE($\mathcal{A}$). The above construction is a log-space reduction from the DATALOG-TUPLE over linear orders with constants to DATALOG-TUPLE over linear orders without constants, mapping the input $(\Pi, P, \bar{a})$ to the instance $(\Pi', P', \bar{c}\bar{a})$, increasing m_L and m_R by r and the total program size by a linear factor. For this constant free version of DATALOG-TUPLE, Theorem 5.18 shows us how to solve it, calculating the type sets introduced in Lemma 5.5. □

No new structure representation issues arise here, since we simply use the solving algorithm for the case without constants here, including representation considerations.

We can also use these type sets computed in the above proof for solving DATALOG-NONEMPTINESS($\mathcal{A}$) on $\mathcal{A} = (A, <, c_1, \ldots, c_r)$:

Corollary 5.30
DATALOG-NONEMPTINESS($\mathcal{A}$) on linear orders $\mathcal{A} = (A, <, c_1, \ldots, c_r)$ with finitely many constants (which may be used by the datalog programs) is decidable.

Proof: On input (Π, P), we first calculate types sets for the modified version Π', in which the constants have been replaced by the variables $C_1, \ldots, C_r$ as above. Then we instantiate the variables $C_1, \ldots, C_r$ by the constants of $\mathcal{A}$ in all types in $\Theta(P', \infty)$

for the IDB P' of Π' corresponding to IDB P of Π, instantiating each C_i by $c_i^{\mathcal{A}}$. Types which are then no longer satisfiable are deleted from the set. If and only if there are satisfiable types remaining, $P_\infty^{\Pi,\mathcal{A}}$ is nonempty. □

However, on dense linear orders we may do better and match the EXPTIME lower bound:

Corollary 5.31
DATALOG-NONEMPTINESS($\mathcal{A}$) and DATALOG-TUPLE($\mathcal{A}$) on dense linear orders $\mathcal{A} = (A, <, c_1, \ldots, c_r)$ with finitely many constants (which may be used by the datalog programs) can both be decided in exponential time.

Proof: This result is a straightforward combination of the preceeding proof and Theorem 5.26. □

Note that even though we use the decidability of the datalog tuple problem to prove the decidability of DATALOG-NONEMPTINESS($\mathcal{A}$) for linear orders with constants, we do not need to make any effectivity assumptions on the linear order $\mathcal{A}$ here. The reason is that the constants are fixed in advance as part of the structure and thus all information about them can be hardwired into the algorithm.

A different approach discussed later, in Chapter 6, leads to an EXPTIME upper bound for DATALOG-NONEMPTINESS on arbitrary linear orders with constants, see Corollary 5.32, stating:

Corollary 5.32
DATALOG-NONEMPTINESS($\mathcal{A}$) on linear orders $\mathcal{A} = (A, <, c_1, \ldots, c_r)$ with finitely many constants (which may be used by the datalog programs) can be decided in exponential time.

Unfortunately, that approach cannot be used to show a singly exponential bound for DATALOG-TUPLE, as we will see in a later chapter.

5.7 Negation in Datalog Rules

When we change the scope from datalog to datalog$^\neg$, negations of EDB atoms become possible which will always be of the form $\neg(t_1 < t_2)$ for some variables or constants t_1 and t_2. Considering, that $\neg(t_1 < t_2)$ is equivalent to $(t_2 < t_1) \vee (t_1 = t_2)$, we may transform these negated EDB atoms to non-negated EDBs. Each atom $\neg(t_1 < t_2)$ will be replaced by rules containing the two variants $(t_2 < t_1)$ and $(t_1 = t_2)$. If more than one negation occurs in a rule, all possible combinations have to be created in replacement rules, similar to the approach of the type disjoint program in Lemma 5.12. As demonstrated there, $t_1 = t_2$ for two variables t_1 and t_2 is simulated by replacing all occurrences of t_2 by t_1. For the cases, when t_1 and t_2 are both constants, we may exclude non satisfiable atoms $t_2 < t_1$ or $t_1 = t_2$ while replacing the negations.

The datalog program Π' resulting, when in a datalog$^\neg$ program Π all negations are replaced, may have a size at most exponential in $|\Pi|$, since each negation per rule doubles the number of possible replacement rules in Π'. This raises the upper bounds for the cases we have considered so far, leading to:

Lemma 5.33 *For datalog$^\neg$ the following upper bounds hold:*

- *DATALOG-NONEMPTINESS($\mathcal{A}$) and DATALOG-TUPLE($\mathcal{A}$) for monadic datalog$^\neg$ programs on linear orders $\mathcal{A} = (A, <)$ are in EXPTIME.*
- *DATALOG-NONEMPTINESS($\mathcal{A}$) on linear orders $\mathcal{A} = (A, <)$ is in* 2EXP .
- *DATALOG-TUPLE($\mathcal{A}$) on linear orders $\mathcal{A} = (A, <)$ is decidable.*
- *DATALOG-NONEMPTINESS($\mathcal{A}$) and DATALOG-TUPLE($\mathcal{A}$) on linear orders $\mathcal{A} = (A, <, c_1, \ldots, c_k)$ with finitely many constants are both decidable.*
- *DATALOG-NONEMPTINESS($\mathcal{A}$) and DATALOG-TUPLE($\mathcal{A}$) on dense linear orders without endpoints $\mathcal{A} = (A, <)$ and dense linear orders without endpoints and finitely many constants, $\mathcal{A} = (A, <, c_1, \ldots, c_k)$, are in* 2EXP .

Maybe a clever combination of the negation elimination process and the creation of the type disjoint program version may lead to a better bound. But this would only be useful in the cases, where a reduction of the number of new rules introduced takes place, e.g. when using constants or other EDB atoms in a rule. For example a rule with body $(x < y), \neg(y < x)$ could be translated to $(x < y)$, omitting the $x = y$ case.

5.8 Representations and DATALOG-TUPLE

In Section 2.5 we have investigated problems with the representation of infinite structures. Besides the abstract model of *structure oracles* we have employed in our results of this chapter, we had a closer look at some possible representations of the dense order $\mathcal{Q} = (\mathbb{Q}, <)$ and the discrete order $\mathcal{Z} = (\mathbb{Z}, <)$. We will now have a look at how these results influence the upper bounds derived.

First note that all of the algorithms discussed here, first compute some representation of the IDBs by type sets and in the last stage it is checked, whether the input tuple is contained in the relation, i.e. satisfies one of the types. Thus we only have to consider this last phase of each algorithm and only need to know how the type sets look like.

5.8.1 Monadic Datalog

While the first case of dense orders is completely independent of the structure representation, Lemma 5.9 shows that for a linear order with minimum, the types just describe the distance to this minimum. This could be modelled as linear order

$\mathcal{N} = (\mathbb{N}, <)$, for which similar arguments as for the order $\mathcal{Z}$ apply. In this monadic case, we only need to answer the query $q_d^{\mathcal{N}}(k, 0, a)$ for the input tuple a and a k having a value that is at most polynomial in the input length. Since $q_d^{\mathcal{N}}(k, 0, a)$ can be answered in time linear in the length of k and a, the polynomial running time of the algorithm is not spoiled by this query.

5.8.2 Linear Orders

In the proof of Theorem 5.18, we show that for each given program length there is a computable bound on the number of stages. Hence for each IDB there is a finite number of types describing this IDB and there is also an upper bound on the ranks of the distance types. Altogether only a finite number of distance queries is needed, each with two input elements and a finite distance. In cases with decidable distance queries as in the example $\mathcal{Z} = (\mathbb{Z}, <)$ as introduced in Section 2.5 the problem is decidable.

5.8.3 Dense Linear Orders

The types describing the IDBs of datalog computations over dense orders are order types. For solving DATALOG-TUPLE, some order queries for the order types corresponding to the IDB relation in question have to be checked. In Corollary 5.28 we have shown, that there are exponentially many order queries to be answered to solve the problem, always with tuple entries from the input as arguments. Using the representation of the rational numbers $\mathcal{Q} = (\mathbb{Q}, <)$ from Section 2.5 as an example for a dense order, we see that all of these queries can be answered in time quadratic in its input length. This constitutes an altogether exponential running time bound for the order queries, leading to an EXPTIME bound as in the oracle case.

5.8.4 Linear Orders With Constants

The complexity of DATALOG-TUPLE on linear orders with constants (having some constant interpretation with constant length) is directly inherited from the cases without constants, with the same results for the different representations.

5.8.5 Decidable Structures

While we have considered representations obeying some complexity bound so far, we may also allow a decidable structure, having no fixed time bound, but keeping the representation *and* our queries decidable and having a finite representation for all elements. In all algorithms for DATALOG-TUPLE considered so far, finitely many queries to the structure are carried out in the last phase of the algorithm. If all of these queries are decidable, then clearly the whole algorithm for DATALOG-TUPLE always terminates and DATALOG-TUPLE is decidable.

5.9 Summary of this Chapter

In this chapter we introduced the concept of types and distance types for datalog programs over linear orders and showed in Lemma 5.5, that for each stage of the fixed point computation of a datalog program, there is a finite set of types describing the stage of the IDB relation. Unfortunately, this does not lead to a finite set for the fixed point.

We applied the concept to show a PTIME upper bound for DATALOG-NONEMPTINESS and DATALOG-TUPLE in some simple cases, like over non-strict orders and for monadic datalog programs.

We then defined the notion of type disjoint programs and showed that for all datalog programs over orders, there is a (possibly exponentially larger) type disjoint version. In Lemma 5.14, we showed, that there is an initialization sequence to solve DATALOG-NONEMPTINESS for type disjoint programs in polynomial time, leading to an EXPTIME upper bound for the nonemptiness of arbitrary datalog programs on linear orders.

From this solution of DATALOG-NONEMPTINESS, we derived a computable uniform upper bound for the number of stages of a datalog program run on linear orders, showing the boundedness of datalog on linear orders and leading to the decidability of DATALOG-TUPLE on linear orders.

Using the algorithm involved as an example for computing with types, we showed a singly exponential bound for DATALOG-TUPLE on dense linear orders, exploiting the quantifier elimination dense orders allow.

Then we had a short look at datalog with constants showing the decidability for both problems. In Chapter 6, we will show a singly exponential bound for DATALOG-NONEMPTINESS on linear orders with constants with different methods.

We had a short look at datalog$^\neg$ which introduces negation of EDB atoms, and we have shown that adding a singly exponential level to the complexity, we can solve this form of datalog with our methods.

As conclusion of the chapter, we studied the influence of the representation of the structure on the algorithms solving DATALOG-TUPLE. The complexity of the algorithms for DATALOG-NONEMPTINESS is (as in the other chapters) independent of the representation of the structure.

Open problems

Question 5.1 *Is there an explicit uniform bound for the number of stages of datalog programs on orders?*

A modification of the proof of Dickson's Lemma studied in [HMS06] for c-bounded runs (see Definition 5.20) could lead to such a bound.

Showing an explicit bound would immediately answer the following question:

Question 5.2 *Can the gap between the computable uniform upper bound and the EXPTIME lower bound for DATALOG-TUPLE be narrowed or even closed?*

Question 5.3 *Is there a more efficient way to solve the problems for datalog$^\neg$?*

Chapter 6

Datalog on Colored Orders

In the finite case, colored orders play an important role for the purely relational representation of strings. A finite order $(B, <, (P_a)_{a\in\Sigma})$ with a finite collection of disjoint monadic relations $(P_a)_{a\in\Sigma}$ is used to represent a string w over alphabet Σ, where B is used as an index set: For each position $b \in B$ there is exactly one membership $b \in P_a$, stating that the b-th character of w is an a. Details of these *word models* can be found in [EF99] and an example in Example 6.1.

Example 6.1 *Let $w = 101011$ be a string over the alphabet $\Sigma = \{0, 1\}$. As a relational structure, this string can be represented by $\mathcal{S} = (\{1, \ldots, 6\}, <^2, P_0^1, P_1^1)$. The relation symbol $<$ is interpreted as the usual order relation on the index set $S := \{1, \ldots, 6\}$, while $P_0 = \{2, 4\}$ and $P_1 = \{1, 3, 5, 6\}$ contain the positions of the occurrences of 0 and 1.*

In a later chapter, in Section 8.3, we use a similar approach to encode strings replacing the order relation by a successor relation.

Besides this extension of this finite formalism to an infinite setting, an application of colored orders is temporal reasoning: The universe serves as ordered set of time instants and the finitely many monadic relations describe the shape of a system under observation at a certain point in time. Using this approach, datalog can be used to describe the behavior of a system.

For example, to describe a chemical process under observation, take the ordered natural numbers as the time instants when an observation takes place. Let M_1 contain all time instants in which the system temperature is above some threshold T and let M_2 contain all the time instants in which a certain reagent occurs in a concentration higher than a threshold. Then the datalog rule $P(y) \leftarrow M_1(x), x < y, M_2(y).$ would set the IDB P to the time instants where the reagent is present in the desired concentration after a certain temperature has been reached. A nonempty IDB P would imply that the reagent can be present in high concentration even if the temperature was high sometime before.

Outline of this Chapter

After the formal definition of colored orders we will discuss how to solve DATALOG-NONEMPTINESS on colored orders. First we will introduce a special variant of colored orders, the interval sequence orders and have a look at DATALOG-NONEMPTINESS on these orders. These orders are a main tool for solving DATALOG-NONEMPTINESS for general colored orders. We use the case $\mathcal{A} = (A, <, M, c_1, \ldots, c_\ell)$ to introduce the basic techniques and illustrate the basic ideas. We then extend these techniques for the main result of this chapter, DATALOG-NONEMPTINESS on colored orders $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_\ell)$. A following brief look at datalog$^\neg$ shows how some restricted form of negation may be allowed. In the last section we address the issues of DATALOG-TUPLE in this context.

Related work

To solve DATALOG-NONEMPTINESS on colored orders we will derive a finite set of properties from each colored order to base an algorithm on. This algorithm creates a type disjoint version of the datalog program in which each IDB is sliced into sets of tuples of the same type. In [Kre99] a similar approach was proposed for constraint databases consisting of a dense linear order as background structure and relations defined by quantifier-free first order formulae (with a finite representation). In this setting the approach leads to results on the data complexity of first order queries, also those using a (deterministic or nondeterministic) transitive closure operator, a least fixed point operator (thus including datalog) and a partial fixed point operator.

Our approach does only allow monadic relations additionally to the linear order and covers datalog queries only, but has the advantages that it is not restricted to dense linear orders, but works for arbitrary colored linear orders, and that it allows more complicated relations to be contained in the database, not only those that are representable by a quantifier-free first order formula. One example for such a relation not definable by a quantifier free first order formula is a monadic relation defining an infinite discrete subset of the linear order. Unlike our method presented in this chapter, methods based on properties of dense orders will (in general) not work on such a discrete order.

6.1 Colored Orders

Definition 6.2 ***(Colored order)***

A colored order is a structure $\mathcal{A} = (A, <, M_1, \ldots, M_m)$, where $\mathcal{A}' = (A, <)$ is an ordered structure and $M_1, \ldots, M_m$ are monadic relations.

A colored order with constants $\mathcal{B} = (B, <, M_1, \ldots, M_m, c_1, \ldots, c_\ell)$ is a colored order with additional constant symbols having a fixed interpretation.

6.1.1 Colored Orders and Types

To solve DATALOG-NONEMPTINESS($\mathcal{A}$) on colored orders we will use a new type concept. The naive combination of monadic EDB relations with the distance type concept from Chapter 5 could be of the form (for a structure with two monadic relations relations M_1 and M_2):

> Between $a \in A$ and $b \in A$ (with $a < b$), there have to be
>
> - i_1 elements x with $x \in M_1$
> - i_2 elements x with $x \in M_2$
> - i_3 elements x with $x \in M_1 \cap M_2$

The following Example 6.3 shows why such an approach fails and it demonstrates with a simple program over a structure with two monadic relations M_1 and M_2 how problematic cases may look like.

Example 6.3 *Let $\mathcal{A} = (A, <, M_1, M_2)$ be a structure with $A = \{1, 2, 3, 4, 5, 6\}$, $<^{\mathcal{A}}$ being the usual order, $M_1 = \{1, 3, 6\}$, $M_2 = \{2, 4, 5\}$; see Figure 6.1 for an overview.*

A	1	2	3	4	5	6
M_1	1		3			6
M_2		2		4	5	
Variables in ρ	x	z_1	z_2	z_3	z_4	y
Membership in ρ	M_1	M_2	$\mathbf{M_2}$	$\mathbf{M_1}$	M_2	M_1

Figure 6.1: Structure $\mathcal{A}$ and variable assignment in Π

On this structure we examine the following one rule program Π:

$$\rho: \quad P(x,y) \leftarrow \quad x < z_1,\ z_1 < z_2,\ z_2 < z_3,\ z_3 < z_4,\ z_4 < y,\\ M_1(x), M_2(z_1), M_2(z_2), M_1(z_3), M_2(z_4), M_1(y).$$

As demonstrated with the variable assignments and the monadic relation memberships shown in Figure 6.1, the only variable assignment satisfying the order atoms does not satisfy the monadic membership constraints. Thus $P_\infty^{\mathcal{A},\Pi} = \emptyset$.

On the other hand, if we use a type description similar to the interval types defined above, we would describe the tuples in $(a_1, a_2) \in P_\infty^{\mathcal{A},\Pi}$ by a type expressing:

- $a_1 < a_2$
- $a_1 \in M_1$, $a_2 \in M_1$
- *There is an element from M_1 between a_1 and a_2.*

- *There are three different elements from M_2 between a_1 and a_2.*

Contrary to the rule of Π*, this description can be satisfied on structure* $\mathcal{A}$*, showing that a description by element counts only, is insufficient, but we also need some information about the order of elements in different relations which is incorporated into what we will call an interval sequence order.*

Thus, only counting elements which have to satisfy different monadic memberships, is not enough. We need more detailed information about the order in which these memberships occur. Extending the distance type concept, this could be established by adding some description strings describing the sequence of occurrence of monadic memberships of elements listing the memberships between tuple entries. Using recursive datalog rules these extended types could themselves occur in types describing the tuples of an IDB relation leading to a recursive type concept. But this concept does not admit some straightforward form of variable elimination, like the one we based the distance type concept on. So we will try a different approach.

6.1.2 Interval Types

To apply our type concept to colored orders we will first introduce a type concept for orders with special monadic relations: disjoint intervals. Later we will discuss how each colored order can be transformed into this form without changing DATALOG-NONEMPTINESS for this order.

Definition 6.4 ***(Interval Orders)***

1. *A colored linear order $\mathcal{A} = (A, <, I_1, \ldots, I_m, c_1, \ldots, c_k)$ with monadic relations $I_1, \ldots, I_m$ and constants $c_1, \ldots, c_k$ is an* interval order *if the following conditions are satisfied:*

   (a) *For all $i \in \{1, \ldots, m\}$, I_i is an* interval*. Formally, for all $a_1, a_2 \in I_i$ with $a_1 < a_2$ and all $b \in A$ the following implication holds:*
   If $a_1 < b < a_2$, then $b \in I_i$.

   (b) *For all $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, k\}$, it holds that*

   $$c_j \notin I_i \ .$$

   (c) *For all $i, i' \in \{1, \ldots, m\}$ with $i < i'$, I_i and $I_{i'}$ are disjoint: $I_i \cap I_{i'} = \emptyset$.*

   (d) *A is* partitioned into these intervals and constants*:*

   $$A = \bigcup_{i=1}^{m} I_i \cup \{c_j \mid j = 1, \ldots, k\}$$

2. *Let $\mathcal{A} = (A, <, I_1, \ldots, I_m, c_1, \ldots, c_k)$ be an interval order. We extend the order $<$ to intervals by:*

(a) For each $a \in A$ and $i \in \{1, \ldots, m\}$, we let

$$\begin{aligned} a < I_i \quad &:\Longleftrightarrow \quad \forall x \in I_i \quad a < x \\ I_i < a \quad &:\Longleftrightarrow \quad \forall x \in I_i \quad x < a \end{aligned}$$

(b) For all $i, j \in \{1, \ldots, m\}$, we let

$$I_i < I_j \quad :\Longleftrightarrow \quad \forall x \in I_i \quad x < I_j$$

Then $<$ is also a linear order on the set $\{I_1, \ldots, I_m\}$ of intervals.

3. *An interval order $\mathcal{A} = (A, <, I_1, \ldots, I_m, c_1, \ldots, c_k)$ is an* interval sequence order *if the following conditions hold:*

$$\begin{aligned} c_1 < c_2 < \cdots < c_k \\ I_1 < I_2 < \cdots < I_m \end{aligned}$$

Intuitively, an interval order is an order with its universe partitioned into its constants and monadic relations which have to be disjoint intervals. Note that all interval orders are colored orders, so all results for colored orders also hold for interval orders.

By the extension of the order to intervals and elements and pairs of intervals in the definition above, comparisons like "all elements x greater than interval I" become meaningful and simplify the notation. Interval sequence orders have the nice property that the constants and the intervals are each ordered ascendingly corresponding to the indexes of their constant (or interval) symbols. By reordering the constant and interval symbols in the vocabulary, each interval order can be transformed to an interval sequence order.

Definition 6.5 ***(Interval Types)***
Let $\mathcal{A} = (A, <, I_1, \ldots, I_m, c_1, \ldots, c_k)$ be an interval sequence order.

1. *An* interval atom *is a formula $x \in I$, where x is a variable and $I \in \{I_1, \ldots, I_m\}$.*

2. *A* constant atom *is a formula $x = c$, where x is a variable and $c \in \{c_1, \ldots, c_k\}$.*

3. *An* interval order type *over variables $\{x_1, \ldots, x_n\}$ is a tuple $\eta = (\gamma, \delta)$, where $\gamma \in \Delta_1(x_1, \ldots, x_n)$ is an order type and δ a set of interval and constant atoms, where for each variable $x \in \{x_1, \ldots, x_n\}$ there is at most one interval- or constant-atom containing x. We denote the set of interval order types over variables $\{x_1, \ldots, x_n\}$ by $\Xi(x_1, \ldots, x_n)$.*

4. *An interval order type $\eta = (\gamma, \delta) \in \Xi(x_1, \ldots, x_n)$ is satisfied by a tuple $\bar{a} = (a_1, \ldots, a_n) \in A^n$, if the entries of $\bar{a}$ satisfy γ and all atoms in δ.*

5. *A* complete interval order type $\eta = (\gamma, \delta) \in \Xi(x_1, \ldots, x_n)$ *is an interval order type satisfying:*

   (a) δ *contains for each variable* $x \in \{x_1, \ldots, x_n\}$ *an interval atom or a constant atom.*
   By $\delta(x)$ *we denote the interval or the constant symbol in this atom.*

   (b) γ *contains the following atoms: For each pair* (x_i, x_j) *of variables* $(1 \leq i < j \leq n)$*, with* $\delta(x_i) = \delta(x_j) = I_\ell$ *for some* $\ell \in \{1, \ldots, m\}$*,* γ *contains one of the atoms*

$$(x_i < x_j), \qquad (x_i = x_j), \qquad (x_j < x_i) \ .$$

6. *We denote the set of all complete interval oder types over variables* $\{x_1, \ldots, x_n\}$ *by* $\Upsilon(x_1, \ldots, x_n)$*.*

7. *The* complete order type $\gamma' \in \Delta_1(x_1, \ldots, x_n)$ induced by *a complete interval order type* $\eta = (\gamma, \delta) \in \Upsilon(x_1, \ldots, x_n)$ *is the complete order type* γ' *with* $\gamma \subseteq \gamma'$ *and which for each pair of variables* (x_i, x_j) *with* $1 \leq i < j \leq n$ *contains an atom*

$$\begin{aligned} &(x_i < x_j), \quad \textit{if } \ \delta(x_i) < \delta(x_j) \\ &(x_j < x_i), \quad \textit{if } \ \delta(x_j) < \delta(x_i) \end{aligned}$$

The last part of this definition justifies, why a complete interval order type $\eta = (\gamma, \delta)$ is called complete, even if γ does not contain an order atom for each pair of variables: The missing order atoms are induced by the interval sequence and the corresponding order on the intervals and constants.

Example 6.6 *We consider the following interval order* $\mathcal{A}$*, which is the restriction of the real numbers to the interval* $[0, 10]$*:* $\mathcal{A} = (A, <, I_1, I_2, c_1, c_2)$ *with* $A = [0, 10] \subset \mathbb{R}$*,* $c_1 = 0$*,* $c_2 = 10$*,* $I_1 = (0, 5]$ *and* $I_2 = (5, 10)$*.*

Then $\mathcal{A}$ *is an interval sequence order with*

$$I_1 < I_2 \qquad \textit{and} \qquad c_1 < c_2 \ .$$

We have a look at $\eta_1 = (\gamma_1, \delta_1) \in \Xi(x_1, x_2, x_3)$ *with:*

$$\begin{aligned} \gamma_1 &= \{x_1 < x_2, x_2 < x_3\} \\ \delta_1 &= \{x_1 \in I_1,\ x_2 \in I_1\} \end{aligned}$$

This interval order type is not *complete, since there is no atom containing* x_3 *in* δ_1*.*

The following similar type $\eta_2 = (\gamma_2, \delta_2) \in \Xi(x_1, x_2, x_3)$ *with*

$$\begin{aligned} \gamma_2 &= \{x_1 < x_2\} \\ \delta_2 &= \{x_1 \in I_1,\ x_2 \in I_1,\ x_3 = c_2\} \end{aligned}$$

however, is complete*, since it contains an atom in δ_2 for each variable and for each pair of variables in the same interval, i.e. $\delta_2(x_1) = \delta_2(x_2) = I_1$, γ_2 contains an order atom.*

The induced complete order type γ_2' of η_2 is:

$$\gamma_2' = \{x_1 < x_2,\, x_2 < x_3,\, x_1 < x_3\}$$

After defining some form of order types for interval sequence orders, we have a look at a distance type concept to be able to describe datalog computations by types. Instead of labeling the atoms as distance atoms, as in Chapter 5, we define a function λ which replaces this labeling and assigns each order atom a vector of distances, one distance per interval. Intuitively, this vector states how many intermediate elements in each interval have to exist to make the type satisfiable.

Definition 6.7 ***(Interval Distance Types)***
Let $\mathcal{A} = (A, <, I_1, \ldots, I_m, c_1, \ldots, c_k)$ be an interval sequence order.

1. *Let $\eta = (\gamma, \delta)$ be a complete interval order type, let γ' be the complete order type induced by η and let $\gamma'' \subseteq \gamma'$ be the set of all inequality atoms in γ'. Let λ be a function mapping inequality atoms to vectors of nonnegative numbers,*

$$\lambda : \gamma'' \to \mathbb{N}^m \ .$$

Then $\kappa = (\eta, \lambda)$ is called an interval distance type *and the set of all interval distance types over variables $\{x_1, \ldots, x_n\}$ is denoted by $\Lambda(x_1, \ldots, x_n)$.*

2. *A tuple $\bar{a} = (a_1, \ldots, a_n) \in A^n$ satisfies $\kappa = (\eta, \lambda) = ((\gamma, \delta), \lambda)$ with γ'' as defined above, if*

   (a) *$\bar{a}$ satisfies η* ***and***

   (b) *for each atom $(x_i < x_i') \in \gamma''$ and $j \in \{1, \ldots, m\}$ with $(\lambda(x_i < x_{i'}))_j > 0$, there exist $b_1, b_2, \ldots, b_{(\lambda(x_i < x_{i'}))_j} \in I_j$ with*

$$a_i < b_1 < b_2 < \cdots < b_{(\lambda(x_i < x_{i'}))_j} < a_{i'}$$

Example 6.8 *We continue Example 6.6: Let $\kappa_1 = (\eta_2, \lambda_1)$ with η_2 be as above. The induced complete order type γ_2' of η_2 does not contain equality atoms, hence $\gamma_2'' = \gamma_2'$.*
We let

$$\lambda(t) = \begin{cases} (2,0), & \text{if} \quad t = (x_1 < x_2) \\ (0,0), & \text{if} \quad t = (x_1 < x_3) \\ (0,0), & \text{if} \quad t = (x_2 < x_3) \end{cases}$$

Then κ_1 is satisfied by $\bar{a} = (1, 4)$, using $b_1 = 2$ and $b_2 = 3$ in Definition 6.7 for the atom $t = (x_1 < x_2)$, as elements in I_1 satisfying $a_1 < b_1 < b_2 < a_2$.

On the other hand, $\kappa_2 = (\eta_2, \lambda_2)$ with $\lambda_2(x_1 < x_2) = (0, 1)$ (and $\lambda_2(t) = (0, 0)$ for $t \neq (x_1 < x_2)$), is not satisfiable: For a satisfying assignment there would have to be an element b_1 in I_1, by the interval atoms containing x_1 and x_2, and this element would have to be from I_2 by λ_2, which is a contradiction since the intervals are disjoint.

Remark 6.9 *Let $\mathcal{A} = (A, <, I_1, \ldots, I_m, c_1, \ldots, c_k)$ be an interval sequence order. Then for each n, the set $\Lambda(x_1, \ldots, x_n)$ can be partially ordered:*

Let $\kappa_1, \kappa_2 \in \Lambda(x_1, \ldots, x_n)$ with $\kappa_1 = (\eta, \lambda_1)$ and $\kappa_2 = (\eta, \lambda_2)$. Then we let $\kappa_1 \preceq \kappa_2$ if and only if for all atoms $t = (x_i, x_i') \in \gamma$ and all $j = 1, \ldots, m$:

$$(\lambda_1(t))_j \leq (\lambda_2(t))_j$$

The following implication similar to Lemma 5.3 can be derived directly from the definition:

For all $\kappa_1, \kappa_2 \in \Lambda(x_1, \ldots, x_n)$ with $\kappa_1 \preceq \kappa_2$ and all tuples $\bar{a} \in A^n$ the following holds:

If $\bar{a}$ satisfies κ_2, then $\bar{a}$ satisfies κ_1.

With these tools, we are ready to describe computations of datalog programs on interval sequence orders, similar to Lemma 5.27. In this chapter we will only consider type disjoint programs on interval sequence orders which we will define first:

Definition 6.10 ***(Interval Order Type Disjoint Program)***

A datalog program Π over an interval sequence order $\mathcal{A}$ is interval order type disjoint *if for each IDB P of Π there exists a complete interval order type η, such that for all tuples $\bar{a} \in A^{\mathrm{ar}(P)}$ it holds that*

$$\bar{a} \in P_\infty^{\Pi,\mathcal{A}} \quad \Longrightarrow \quad \bar{a} \text{ satisfies } \eta \ .$$

In the following lemma we show the connection between interval distance types and datalog computations. The methods used in this proof are a generalization of the proof of Lemma 5.5 and the proof is presented here to justify the creation of an initialization sequence to solve DATALOG-NONEMPTINESS.

Lemma 6.11 *Let Π be an interval order type disjoint program over an interval sequence order $\mathcal{A} = (A, <, I_1, \ldots, I_m, c_1, \ldots, c_k)$. Then for each $i > 0$ and each IDB P of Π there exists a finite set $\Theta(P, i) \subset \Lambda(x_1, \ldots, x_{\mathrm{ar}(P)})$ such that for all $\bar{a} \in A^{\mathrm{ar}(P)}$ it holds that*

$$\bar{a} \in P_i^{\Pi,\mathcal{A}} \quad \Longleftrightarrow \quad \text{there is a } \kappa \in \Theta(P, i) \text{ such that } \bar{a} \text{ satisfies } \kappa \ .$$

Proof: We prove this lemma by induction on the stages, similar to Lemma 5.27. Parallel to the proof of this lemma we give a brief overview of the notations and ideas in Example 6.12.

For the **induction base** assume that a rule ρ is applied containing only EDBs in the body. Following the type disjoint form of the program Π, all satisfying assignments of ρ have to satisfy some fixed complete interval order type $\eta = (\gamma, \delta)$ and its induced complete order type γ'. Using this type and the EDB atoms of ρ we build a directed graph $G = (V, E)$ with vertex labels.

The vertex set of G is the set of variables of ρ with the exception that for x_i, x_j with $(x_i = x_j) \in \gamma'$ we create only one vertex for these variables.

The edge set is created from the EDBs atoms of ρ and the order atoms of γ': For each atom $x_i < x_j$, we add an edge (x_i, x_j). The vertex labels $\psi : V \to \{I_1, \ldots, I_m, c_1, \ldots, c_k\}$ are defined from the atoms of the form $I_i(x_j)$ in ρ and δ.

If a conflict arises during this construction, then there is no valid instantiation of rule ρ, which contradicts our assumption. After creating this graph G there may be vertices $\{z_1, \ldots, z_n\}$ on which ψ is not defined and we have yet to assign labels to these vertices. Since there could be more than one way of assigning these labels to get a satisfiable type, we consider copies $\{G_1, \ldots, G_{n^{m+k}}\}$ of the graph G, one for each assignment of intervals and constants $\{I_1, \ldots, I_m, c_1, \ldots, c_k\}$ to the variables $\{z_1, \ldots, z_n\}$.

For each graph G_j we check if G_j is consistent, where G_j is inconsistent if a directed cycle exists in G_j or if an edge (v_1, v_2) exists in G_j with $\psi(v_1) > \psi(v_2)$ which would imply variables assigned to constants or values in intervals contradicting the order of these objects. We remove G_j if it is inconsistent.

Now each remaining G_j is consistent and leads to a set of variable assignments being instantiations of the rule ρ which we encode in an interval distance type κ_j:

For each pair of rule head variables $(x_i, x_{i'})$ for which a directed path p from x_i to $x_{i'}$ exists in G_j, there is also an atom $x_i < x_{i'}$ in γ', since γ' is complete. We define $\lambda(x_i < x_{i'})$ by considering the set of all directed paths $D := \{p_1, \ldots, p_\ell\}$ from x_i to $x_{i'}$ in G_j. For each $\nu \in \{1, \ldots, m\}$ and each $p \in \{p_1, \ldots, p_\ell\}$ we count the number of internal vertices on p which are labeled by I_ν. and denote this number by $n(p, \nu)$. For each I_ν we choose the path with the maximal value $n(p, \nu)$ to define $\lambda(x_i < x_{i'})$:

$$\lambda(x_i < x_{i'}) := (\max_{p \in D}\{n(p, 1)\}, \ldots, \max_{p \in D}\{n(p, m)\})$$

Now assume that a tuple $\bar{a}$ satisfies $\kappa_j = (\eta, \lambda_j)$ and hence satisfies the complete interval type η of IDB P. For each pair of tuple entries $(a_i, a_{i'})$ with $(x_i, x_{i'}) \in \gamma'$ (recall, that this is the complete order type induced by η and also satisfied by $\bar{a}$), there exist $(\lambda_j(x_i < x_{i'}))_\nu)$ pairwise different elements in I_ν between a_i and a_i'. Collecting these elements for all atoms in γ', they can be used for an instantiation of the rule ρ.

If on the other hand we have an instantiation $\bar{a}$ of the rule ρ for head variables $\bar{x}$ and $\bar{b}$ for the variables $\bar{y}$ occurring in the body only then $\bar{a}$ satisfies $\eta = (\gamma, \delta)$ and hence also the induced complete order type γ'. Each of the tuple entries of $\bar{b}$ will be in some interval or be equal to a constant and we may choose the graph G_j with the coinciding vertex labeling. Since the tuples $\bar{a}$ and $\bar{b}$ lead to a satisfying assignment of the variables, G_j may not be inconsistent, hence there will be a type κ_j satisfied by $\bar{a}$.

In the **induction step** we have a rule ρ with some head IDB P and a body containing both EDB and IDB atoms, which is applied in stage $i-1$ to get stage i. We let $\Theta(P, i) = \Theta(P, i-1)$ and create some types which are added to $\Theta(P, i)$ using rule ρ.

Since the program is interval type disjoint, there is a complete interval order type $\eta = (\gamma, \delta)$, which all tuples in P_i^Π have to satisfy for all $i \geq 0$, and a complete order

type γ' induced by η. We denote the IDB occurrences in the body of ρ by $P_1, \ldots, P_\ell$. By induction assumption we have for each of these IDB occurrences P_j a nonempty type set $\Theta(P_j, i-1)$. We now consider each combination $(\tau_1, \ldots, \tau_\ell)$ of types, and denote $((\gamma_j, \delta_j), \lambda_j) = \tau_j$ and by γ'_j the induced complete order type.

For each such $(\tau_1, \ldots, \tau_\ell)$ we create a directed graph $G = (V, E)$ with vertex labels ψ and edge labels σ. The vertices of G are the variables of ρ, where as above only one vertex is used for variables $x_i, x_{i'}$ with $(x_i = x_{i'}) \in \gamma'$. The edges are created corresponding to the EDB atoms of ρ (an edge $(x_i, x_{i'})$ for each atom $x_i < x_{i'}$), the order type γ', and all order types γ'_j.

The edge labels $\sigma : V \to \mathbb{N}^m$ are used to collect the distance information from all interval distance types: For each edge $e = (x_i, x_{i'}) \in E$, let $L_e \subset \{1, \ldots, \ell\}$ be the set of $\nu \in \{1, \ldots, \ell\}$ with $(x_i < x_{i'}) \in \gamma'_\nu$. Informally, the set L_e contains all the indices of the types from $\{\tau_1, \ldots, \tau_\ell\}$ with some distance information about the endpoints of this edge. In each interval I_j we determine the maximal distance for this edge

$$d_j(e) = \max_{j \in L_e} \{(\lambda(x_i < x_{i'}))_j\}$$

and label the edge with a vector containing all these maximums:

$$\sigma(e) = (d_1(e), \ldots, d_m(e)) \ .$$

For each edge $e = (x_i, x_{i'})$ the label $\sigma(e)$ gives us some information that is common to all satisfying assignments to the variables: If a_i is assigned to x_i and $a_{i'}$ to $x_{i'}$ in a satisfying assignment, then $a_i < a_{i'}$ and for each $j = 1, \ldots, m$ there have to be $(\sigma(e))_j$ different elements in I_j between a_i and $a_{i'}$ to satisfy all types $\tau_1, \ldots, \tau_\ell$ simultaneously.

As last part of the graph G we define a vertex labeling ψ which is analogous to the one in the induction base: We label each vertex with either an interval or a constant, according to δ and $\delta_1, \ldots, \delta_\ell$. If there is an inconsistency in this step, we drop this combination $(\tau_1, \ldots, \tau_\ell)$ of types and carry on with the next one.

After this process, there may be vertices $\{z_1, \ldots, z_n\}$ which are still unlabeled. As in the induction base we create $(m+k)^n$ copies of G, each with a different labeling of these vertices with an interval or a constant.

Since only a consistent graph may lead to a satisfiable type, each graph G' is now checked for consistency:

- If there is a directed cycle in G' then G' is inconsistent, since variables x on this cycle will have to satisfy $x < x$, which is impossible.
- For each edge we check if the endpoints are labeled consistently: Let $e = (x, y)$ be an edge of G'. Then x and y are labeled consistently if
  - x and y are labeled with the same interval, i.e. $\psi(x) = \psi(y) = I_j$ for some j, **or**
  - the labels obey the order relation $x < y$, i.e. $\psi(x) < \psi(y)$.

- For each vertex we check if the labels of the ingoing edges are consistent with the vertex label: Let x be a vertex of G' with label $\psi(x)$ and let $e = (y, x)$ be an ingoing edge. If there is a $j \in \{1, \ldots, m\}$ with $(\sigma(e))_j > 0$ and $\psi(x) < I_j$ then G' is inconsistent: In this case for a satisfying variable assignment assigning some value a to x, there would have to be an element $b \in I_j$ with $b < a$, since b has to be between the values assigned to y and x, and following from $b \in I_j$ and $\psi(x) < I_j$ also $a < b$ would hold, a contradiction.

- We run an analogous procedure for all outgoing edges for each vertex.

By construction, each vertex is labeled with an interval and a constant. Intervals and constants can be ordered linearly in an interval sequence order and we can use the transitivity to restrict our consistency check to edges only.

From each remaining graph $G = (V, E)$ with labels ψ and σ, a interval distance type $\kappa_G = ((\gamma, \delta), \lambda_G)$ is created and added to $\Theta(P, i)$:

For each pair of variables $(x_i, x_{i'})$ in G, such that the set of paths from x_i to $x_{i'}$, $W_G(x_i, x_{i'})$ is nonempty, let for $j = 1, \ldots, m$

$$(\lambda(x_i < x_{i'}))_j = \max_{p \in W_G(x_i, x_{i'})} \sum_{e \in p} (\sigma(e)_j + n_j(p)) \quad ,$$

where $n_j(p)$ denotes the number of vertices x on path p with $\psi(x) = I_j$.

Intuitively, $(\lambda(x_i < x_{i'}))_j$ counts the number of elements from I_j which occur on the path from x_i to $x_{i'}$, either directly in G, or in the recursively created edge labelings. By construction of the graph, a tuple $\bar{a}$ is satisfying κ_G if and only if there is a variable assignment to the variables occurring only in the body, such that the types used for labeling the edges of the graph are satisfied.

It now remains to show that the tuples in P_i^Π are exactly those which satisfy a type in $\Theta(P, i)$.

- Each $\bar{a} \in P_i^\Pi$ is either contained in some P_j^Π for $j < i$, in which case it satisfies some $\kappa \in \Theta(P, j) \subseteq \Theta(P, i)$ by induction hypothesis, or it is added in stage i. But then $\bar{a}$ is part of an instantiation $(\bar{a}, \bar{b})$ of rule ρ. By induction hypothesis the corresponding tuple entries satisfy some interval distance types for the IDBs $P_1, \ldots, P_\ell$. The entries of tuple $\bar{b}$ assigned to the body variables of ρ each satisfy some interval or constant atom. In the above construction, there will be a graph G corresponding to these interval and order atoms and since we have a satisfying assignment $(\bar{a}, \bar{b})$, this graph will be consistent and lead to an interval distance type κ_G satisfied by $\bar{a}$.

- Let a tuple $\bar{a}$ satisfy one of the types $\kappa \in \Theta(P, i)$. If $\bar{a}$ satisfies a $\kappa \in \Theta(P, i-1)$, then by induction hypothesis, $\bar{a} \in P_{i-1}^\Pi \subseteq P_i^\Pi$. Now let $\bar{a}$ satisfy only a $\kappa \in \Theta(P, i) \setminus \Theta(P, i-1)$. From $\bar{a}$ and $\kappa = (\eta, \lambda)$ we create an instantiation of ρ: For each pair (a_i, a_j) with $a_i < a_j$, we know the number of elements in each interval that exist between a_i and a_j. By the way how κ was created, we know

that there is a directed graph G with edge and vertex labelings as above, which is consistent. For this, the vertices corresponding to the body variables of ρ which are not labeled using the atoms of rule ρ, need to be labeled. By the construction of the function λ, there is a labeling for each such node with an interval or a constant and a type $\tau_j \in \Theta(P_j, i-1)$ for each IDB P_j of rule ρ, such that the graph created is consistent and hence types τ_j are satisfiable with a common variable assignment of ρ.

By induction hypothesis, we know that all variable assignments satisfying each τ_j are included in the corresponding relation $(P_j)^{\Pi}_{i-1}$. Hence the variable assignment satisfying these types leads to an instantiation of rule ρ and therefore $\bar{a} \in P_i^{\Pi}$.

□

Example 6.12 ***(For Lemma 6.11)***

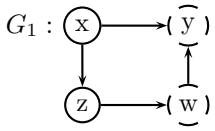
For an example demonstrating the graphs created in the proof of this lemma, we use the interval sequence order defined in Example 6.6: $\mathcal{A} = (A, <, I_1, I_2, c_1, c_2)$ *with* $A = [0, 10] \subset \mathbb{R}$, $c_1 = 0$, $c_2 = 10$, $I_1 = (0, 5]$ *and* $I_2 = (5, 10)$.

On this structure we consider the following datalog program (which is interval type disjoint):

$$P(x, y) \leftarrow I_1(x),\ x < z,\ I_1(z),\ I_2(w),\ w < y,\ I_2(y). \tag{6.1}$$

$$P(x, y) \leftarrow P(x, z),\ z < w,\ w < y,\ I_2(y). \tag{6.2}$$

(6.1) is a non recursive rule and can be considered in the induction base. *From the EDB atoms the following graph is created:*



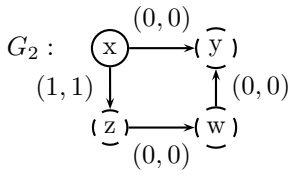
Vertex labels:

○ $= I_1$
◡ $= I_2$

$$\gamma = \emptyset,\ \gamma' = \{x < y\},$$
$$\delta = \{x \in I_1,\ y \in I_2\},$$
$$\lambda_1(x < y) = (1, 1),$$
$$\kappa_{G_1} = ((\gamma, \delta), \lambda_1)$$

This graph G is created from the complete order type γ' induced by the complete interval order type $\eta = (\gamma, \delta)$. From this graph G, the function λ is created, considering the two paths from x to y and using the maximum element count in each interval. Together with η this is the interval distance type κ_{G_1} which additionally to η contains the information that for each tuple (a_1, a_2) satisfying this type, there have to exist $b_1 \in I_1$ and $b_2 \in I_2$ with $a_1 < b_1 < b_2 < a_2$.

For the rule in (6.2) the interval order type is (by coincidence) the same as for the first rule. As recursive rule, we use it to show an example for the graph G_2 created in the induction step *of the proof of the preceeding lemma. Beside vertex labels as above, each edge is labeled by a vector.*

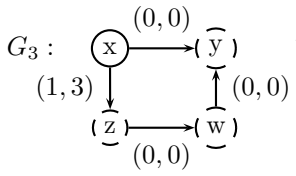


$$\gamma = \emptyset,\ \gamma' = \{x < y\},$$
$$\delta = \{x \in I_1,\ y \in I_2\},$$
$$\lambda_2(x < y) = (1, 3),$$
$$\kappa_{G_2} = ((\gamma, \delta), \lambda_2)$$

Note that the program rule does not constrain w to any interval or constant and this vertex has to be labeled. Since the labels c_1 or I_1 directly lead to an inconsistency because of the label I_2 of z, and because the label I_2 of y does not allow label c_2 for w, the label I_2 is the only remaining label for w.

$\kappa_2(x < y)$ is defined considering the two paths from x to y. While the direct path (the edge (x, y)) has label $(0, 0)$ and no internal vertices, the path over z and w has a total edge label sum of $(1, 1)$ and two internal vertices from interval I_2, which leads to $\lambda_2(x < y) = (1, 1) + (0, 2) = (1, 3)$.

Since the rule in (6.2) is recursive, we may use the type just created in the induction step to derive a new type, where the type created from graph G_2 occurs in the edge labeling of the edge (x, z):



$$\gamma = \emptyset,\ \gamma' = \{x < y\},$$
$$\delta = \{x \in I_1,\ y \in I_2\},$$
$$\lambda_3(x < y) = (1, 5),$$
$$\kappa_{G_3} = ((\gamma, \delta), \lambda_3)$$

But this type does not lead to new tuples in the IDB P since $\kappa_{G_2} \preceq \kappa_{G_3}$.

If $\mathcal{A}$ is an interval sequence order in which the intervals are infinite sets, we may solve DATALOG-NONEMPTINESS($\mathcal{A}$) by creating an initialization sequence, similar to Lemma 5.14.

Lemma 6.13 ***(Initialization Sequence on Interval Sequence Orders)***
Let $\mathcal{A} = (A, <, I_1, \ldots, I_m, c_1, \ldots, c_k)$ be an interval sequence order with infinite intervals, i.e. for $j = 1, \ldots, m$, $|I_j| = \infty$.

Let Π be an interval order type disjoint datalog program over $\mathcal{A}$.

Then there exist an $i_s \leq n_I$ and a sequence $s = (\rho_0, \rho_1, \ldots, \rho_{i_s - 1})$ of rules, such that after applying ρ_i at stage i for $i = 0, \ldots, i_s - 1$, the emptiness is determined, i.e. for all IDBs P of Π it holds that

$$P_{i_s}^{\Pi,\mathcal{A}} = \emptyset \quad \Rightarrow \quad P_{\infty}^{\Pi,\mathcal{A}} = \emptyset\ . \tag{6.3}$$

This sequence s can be computed in time polynomial in $n_R \cdot n_I$.

Proof: We create the sequence s by cycling through the rules n_I times, adding those rules to s which change an empty IDB to nonempty. Formally:

$s = (\rho_0, \rho_1, \ldots, \rho_{i_s-1})$ such that there exist IDBs $P_1, \ldots, P_{i_s}$ with $(P_i)_i^\Pi = \emptyset$, and after applying ρ_i, $(P_i)_{i+1}^\Pi \neq \emptyset$ for $i = 0, \ldots, i_s - 1$.

We continue this process until no more rules can be applied to make an empty IDB nonempty, but this can happen at most n_I times, immediately leading to the time bound for the computation. Note that nonempty IDBs are never modified by s.

The reason why this is sufficient, is a combination of previously shown results: By Lemma 6.11 each stage of an IDB relation can be expressed by a finite set of interval distance types which have all the same underlying interval order type, since the program is interval type disjoint. Let τ be an interval distance type created by a rule application making an IDB nonempty. Because each interval in $\mathcal{A}$ is infinite and the interval distance types are monotone by Remark 6.9, all tuples satisfying this type with arbitrarily high distances between the tuple entries are also satisfying this interval distance type τ, i.e. there will also be tuples satisfying all distance types τ' with $\tau \preceq \tau'$.

Note that if between two constants there are only finitely many elements, then all these elements are labeled with constants and by their representation in the distance type, the information about these finitely many elements is transferred from body IDBs to the rule and head IDB.

So if a rule is applied, then for each body IDB P, there is a distance type τ in the type set of P and we can choose tuples satisfying τ, with distances large enough to find satisfying assignments for the other variables of the rule, which may occur in other EDB or IDB atoms, even those constrained to be between tuple entries of P.

Thus a rule of Π can be instantiated if and only if all body IDBs are nonempty and the interval order types of all head and body IDBs and the EDBs is consistent. This consistency check can be implemented by representing the interval order type of all rule variables as graph and checking it for consistency — as in the proof of Lemma 6.11, but without the distance information and the corresponding edge labels. This check will have running time polynomial in the rule length.

We now show property (6.3) by contradiction:

Let $U = \left\{ R \;\middle|\; R_{i_s}^\Pi = \emptyset \wedge R_\infty^\Pi \neq \emptyset \right\}$ be the set of IDBs changing to nonempty after s and assume $U \neq \emptyset$. Then for each $R \in U$ there exist an $i_R \in \mathbb{N}$ and a rule ρ_R with:

$$R_{i_R}^\Pi = \emptyset, \text{ and applying } \rho_R : \; R_{i_R+1}^\Pi \neq \emptyset \; .$$

Let $P \in U$ be the IDB with $i_P = \min \{ i_R \mid R \in U \}$. By the definition of U and by the choice of i_P, all $Q \in U \setminus \{P\}$ have to satisfy $Q_{i_P}^\Pi = \emptyset$. Since a rule can be applied if and only if all body IDBs are nonempty, the rule ρ_R cannot depend on them and can be applied in stage i_P leading to a sequence of rule applications making more IDBs nonempty, a contradiction to the construction of s. □

In the rest of this chapter we will show how to achieve the conditions to apply this lemma for colored orders.

6.1.3 Modifying Datalog Programs

The general idea in this chapter is to investigate a transformation from the colored order $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ in question to an interval sequence order $\mathcal{B}$ while keeping the solution of DATALOG-NONEMPTINESS($\mathcal{A}$) and DATALOG-NONEMPTINESS($\mathcal{B}$) equivalent. This transformation from $\mathcal{A}$ to $\mathcal{B}$ will consist of only finitely many changes, which can then be used in an algorithm transforming programs on $\mathcal{A}$ to equivalent programs on $\mathcal{B}$. For the generated program, DATALOG-NONEMPTINESS($\mathcal{B}$) can then be solved by the methods introduced in the previous section.

The main idea behind the interval sequence order $\mathcal{B}$ and the transformation from $\mathcal{A}$ to $\mathcal{B}$ is that the memberships of elements in the monadic relation $M_1, \ldots, M_m$ can be translated to memberships in the disjoint intervals.

Remark 6.14 *The general plan for solving DATALOG-NONEMPTINESS($\mathcal{A}$) on colored orders $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ consists of two phases and can be summarized in the following way:*

Phase 1: *Analyze the structure $\mathcal{A}$ and create the solving algorithm for DATALOG-NONEMPTINESS($\mathcal{A}$) by using an interval sequence order $\mathcal{B}$.*

Phase 2: *Use this algorithm for solving problem instances (Π, P) of DATALOG-NONEMPTINESS($\mathcal{A}$).*

The algorithm has two main parts:

1. *Transform the program Π to an interval order type disjoint program Π' according to the changes from $\mathcal{A}$ to $\mathcal{B}$.*
2. *Generate the initialization sequence for Π'.*

Note that for the second phase, we do not need to access the structures $\mathcal{A}$ and $\mathcal{B}$, since all information necessary for solving DATALOG-NONEMPTINESS is transferred to the program Π' by our algorithm.

To transform the colored order $\mathcal{A}$ into an interval sequence order with infinite intervals, we will add finitely many constants in a first stage to ensure that there are no finite intervals left. Then all constants are ordered as in an interval sequence order. To justify this, we introduce two technical lemmas formalizing some intuition about the datalog programs and the change of constants:

Lemma 6.15 *Let $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ be a (colored or non-colored order) with $m \geq 0$, $k \geq 0$ and let Π be a datalog program over $\mathcal{A}$.*

Let $\mathcal{A}' = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k, c_{k+1}, \ldots, c_{k+\ell})$ be an expansion of $\mathcal{A}$ with some additional constants, $\ell > 0$.

Then Π is also a program over $\mathcal{A}'$ and for each IDB P of Π the relations are the same on both structures:

$$P_\infty^{\Pi,\mathcal{A}} = P_\infty^{\Pi,\mathcal{A}'}$$

Proof: Π does not use any of the constant symbols $c_{k+1}, \ldots, c_{k+\ell}$, so all possible computations of Π work exactly as on $\mathcal{A}$. □

Lemma 6.16 *Let $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ be a (colored or non-colored order) with $m \geq 0$, $k \geq 1$ and let Π be a datalog program over $\mathcal{A}$.*

Let $\pi : \{1, \ldots, k\} \to \{1, \ldots, k\}$ be a permutation.

Let $\mathcal{A}_\pi = (A, <, M_1, \ldots, M_m, c'_1, \ldots, c'_k)$ be a colored order with the numbering of the constants derived from $\mathcal{A}$ by applying the permutation π: For $j = 1, \ldots, k$ the interpretations of the constants satisfy $c'_{\pi(j)} = c_j$.

Then the program Π can be translated to a program Π' by replacing each occurrence of each constant c_j by $c'_{\pi(j)}$, for which all relations computed by Π and Π' coincide, i.e. for each IDB P of Π:

$$P_\infty^{\Pi,\mathcal{A}} = P_\infty^{\Pi',\mathcal{A}'}$$

Proof: Since the constants in Π are replaced with constants with the same interpretation as in $\mathcal{A}$, the computations of Π and Π' coincide. □

6.2 Orders with one Monadic Relation and Constants

The easiest case to demonstrate our method for solving DATALOG-NONEMPTINESS on colored orders is the case of an order with one additional monadic relation M and finitely many constants $c_1, \ldots, c_k$ for some k.

Given a colored order $\mathcal{A} = (A, <, M, c_1, \ldots, c_k)$, we derive a finite sequence of transformations from $\mathcal{A}$ to an interval sequence order $\mathcal{B}$ with infinite intervals. From this finite set of transformations we derive a method to transform the input (Π, P) of DATALOG-NONEMPTINESS($\mathcal{A}$) to an interval order type disjoint program Π' on $\mathcal{B}$ for which we know how to solve the nonemptiness of the relations by the results of the preceeding section.

We first partition the colored order $\mathcal{A}$ using the constants. An example for the following definitions is given in Example 6.23.

Definition 6.17 *Let $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ be a linear order ($m = 0$) or a colored linear order ($m > 0$) with $k \geq 0$ and $c_1 < c_2 < \cdots < c_k$. Then the intervals defined by the constants of $\mathcal{A}$ are for $k > 0$:*

- $I_0 := \{a \in A \mid a < c_1\}$
- $I_i := (c_i, c_{i+1}) := \{a \in A \mid c_i < a < c_{i+1}\}$ *for* $i = 1, \ldots, k-1$
- $I_k := \{a \in A \mid c_k < a\}$

In the case $k = 0$, we let $I_0 = A$.

We call $I_0, I_1, \ldots, I_k$ the constant intervals *of the structure $\mathcal{A}$.*

Together with the constants these intervals partition the universe A of the structure $\mathcal{A}$.

For these constant intervals of the structure $\mathcal{A}$, we replace all intervals that contain only finitely many elements or only finitely many elements that are in M:

Definition 6.18 ***(Colored Order with Empty or Infinite Constants Intervals)***

Let $\mathcal{A} = (A, <, M, c_1, \ldots, c_\ell)$ be a colored order with constants, without loss of generality $c_1 < c_2 < \cdots < c_\ell$. Then we define the colored order with empty or infinite constants intervals to $\mathcal{A}$ *as the structure derived from $\mathcal{A}$ by the following steps:*

1. *Create the constant intervals as in Definition 6.17. For each interval I with $0 < |I \cap M| < \infty$, we add a new constant for each element of $I \cap M$.*

2. *For the structure created in the previous step, create the constant intervals and then for each interval I with $0 < |I| < \infty$, add a new constant for each element of I.*

The resulting structure is then of the form $\mathcal{A}' = (A, <, M, c'_1, \ldots, c'_{\ell'})$, where we may again assume $c'_1 < \cdots < c'_{\ell'}$.

This transformation has no impact on the nonemptiness of relations:

Lemma 6.19 *Let $\mathcal{A}$ be a colored order with constants and $\mathcal{A}'$ the* colored order with empty or infinite constant intervals to $\mathcal{A}$. *Let Π be a datalog program over $\mathcal{A}$ and P an IDB of Π. Then $P_\infty^{\Pi,\mathcal{A}} = \emptyset$ if and only if $P_\infty^{\Pi,\mathcal{A}'} = \emptyset$.*

Proof: Since we only add finitely many constants in the transformation from $\mathcal{A}$ and $\mathcal{A}'$ and change their ordering, the result follows directly from Lemma 6.15 and Lemma 6.16. □

If we consider the constant intervals of the structure $\mathcal{A}'$, then each constant interval I has one of the following shapes:

1. $I = \emptyset$
2. $I \cap M = \emptyset$ and $|I| = \infty$
3. $|I \cap M| = \infty$

If we create an interval sequence order from $\mathcal{A}'$ and omit the empty intervals, the constants and the nonempty intervals form a partition of the universe A as desired. However, the intervals of the third shape may contain elements of M and elements not in M. To solve DATALOG-NONEMPTINESS without considering monadic memberships, we will transform the structure into a shape in which each such interval contains only elements from M. We achieve this by omitting all elements *not* in M from each such interval. Formally the transformation, works as follows:

Definition 6.20 ***(Colored Order With Empty or Uniform Infinite Constant Intervals)***

Let $\mathcal{A}' = (A, <, M, c'_1, \ldots, c'_{\ell'})$ be a colored order with empty or infinite constant intervals. We transform it to a colored order with empty or uniform infinite constant intervals *$\mathcal{A}'' = (A', <, M, c'_1, \ldots, c'_{\ell'})$ by the following transformation. Create the constant intervals as in Definition 6.17.*

For each interval I with $I \cap M \neq \emptyset$ and $I \setminus M \neq \emptyset$, we replace I by $I \cap M$, thus removing $I \setminus M$ from A'.

Using this transformation, we have changed all infinite intervals containing both elements in M and not in M. That the nonemptiness of IDB relations has not been changed is not obvious and remains to show:

Lemma 6.21 *Let $\mathcal{A}' = (A, <, M, c_1, \ldots, c_\ell)$ be a colored order with empty or infinite constant intervals. Let $\mathcal{A}''$ be the colored order with empty or uniform infinite constant intervals as defined in the previous definition. Let Π be a datalog program over $\mathcal{A}'$ and P an IDB of Π. Then it holds that*

$$P_\infty^{\Pi,\mathcal{A}'} \neq \emptyset \quad \Longleftrightarrow \quad P_\infty^{\Pi,\mathcal{A}''} \neq \emptyset$$

Proof: One direction of the implication is straightforward: Since $\mathcal{A}''$ is a substructure of $\mathcal{A}'$, each IDB relation nonempty on $\mathcal{A}''$ will also be nonempty on $\mathcal{A}'$ by 2.4. Now we need to show the other direction of this implication.

The main observation used for this proof is that since datalog is a positive existential formalism, a datalog program is not able to force some of the tuple entries of a satisfying assignment to be *not* contained in M. So if there are enough elements from M in each interval, we may simply replace the tuple entries not in M by entries from M. More formally, we show the following claim by induction on the stages:

> Let ρ be the rule applied in stage i during the computation of Π on $\mathcal{A}'$ and let $\bar{a}\bar{b}$ be an instantiation of rule ρ in stage i, where $\bar{a}$ is the tuple assigned to the head variables. Let s be the sequence of rules in stages 0 to $i-1$ whose instantiations are needed for the stage i; let $\bar{d}$ be a tuple consisting of the concatenation of these instantiations. Let τ be a distance type of the tuple $\bar{c}\bar{d}\bar{a}\bar{b}$, where $\bar{c}$ are the elements denoted by the constants $c_1, \ldots, c_\ell$.
>
> Then there is an instantiation $\bar{a}'\bar{b}'$ of rule ρ in stage i on $\mathcal{A}''$ and there are instantiations of the rules in s on $\mathcal{A}''$, concatenated into a tuple $\bar{d}'$ such that the tuple $\bar{c}\bar{d}'\bar{a}'\bar{b}'$ satisfies τ. Entries of $\bar{d}\bar{a}\bar{b}$ occurring in some unchanged constant intervals are equal to the corresponding entries of $\bar{d}'\bar{a}'\bar{b}'$.

Induction base: Rule ρ only contains EDB atoms in the body, i.e. order atoms and atoms of the form $M(x)$. $\bar{d}$ and $\bar{d}'$ are empty. From $\bar{a}\bar{b}$ we create $\bar{a}'\bar{b}'$: For each unchanged interval we copy the elements from $\bar{a}\bar{b}$.

Let I be a constant interval that differs between $\mathcal{A}'$ and $\mathcal{A}''$. Since I is infinite, we may choose the corresponding elements for $\bar{a}'\bar{b}'$ to satisfy the order atoms of ρ and the distance type τ (together with the elements from the other intervals and the constants $\bar{c}$, which are all greater or smaller than all elements of I). After we have chosen elements satisfying τ for all changed intervals, the tuple $\bar{c}\bar{a}'\bar{b}'$ will satisfy τ. All entries of $\bar{a}'\bar{b}'$ in the changed intervals are also from M, so each atom of the form $M(x)$ in ρ will also be satisfied, hence $\bar{a}'\bar{b}'$ is an instantiation of ρ on $\mathcal{A}''$.

Induction step: Here a rule ρ is instantiated over $\mathcal{A}'$ containing both IDB and EDB atoms in the body.

In this rule, tuples from IDB relations are used to create the instantiation $\bar{a}\bar{b}$. These tuples are from instantiations of rules in earlier stages and hence contained in $\bar{d}$. With the IDB occurrences in rule ρ some of these elements are assigned to the rule variables and become part of the instantiation $\bar{a}\bar{b}$, hence certain entries of $\bar{d}$ have to be equal to some entries in $\bar{a}\bar{b}$.

We describe the rule instantiation by a distance type: If two entries s and t of $\bar{c}\bar{d}\bar{a}\bar{b}$ are assigned to the same variable, we add an atom $s = t$ to τ. If there is an order atom $x < y$ in this rule, and the corresponding entries of $\bar{c}\bar{d}\bar{a}\bar{b}$ are s and t, we add an order atom $s < t$ to τ. We then compute the transitive closure of τ, where we sum order atoms to distance atoms, i.e. $s < t$ and $t < u$ would lead to the distance atom $s <_2 u$.

Note that then entries of the tuple $\bar{d}\bar{c}\bar{a}\bar{b}$ from different IDB occurrences and hence different rule instantiations are then forced to be equal, if they are assigned to the same variable in ρ.

Let τ' be the restriction of τ to $\bar{c}\bar{d}$ which is created by removing the atoms containing occurrences of entries of $\bar{a}\bar{b}$. $\bar{c}\bar{d}$ satisfies τ', and τ' is satisfiable if and only if τ is satisfiable.

By induction hypothesis on the last rule instantiation in s, we know that there is a tuple $\bar{d}'$ containing the instantiations of the rules in s such that $\bar{c}\bar{d}'$ satisfies τ'. From this we can create a tuple $\bar{c}\bar{d}'\bar{a}'\bar{b}'$ satisfying τ where in constant intervals of $\mathcal{A}'$ that are unchanged in $\mathcal{A}''$, we may use the same values as in $\mathcal{A}'$.

Then $\bar{a}'\bar{b}'$ is an instantiation of ρ over $\mathcal{A}''$:

- The variables of ρ occurring in some IDB atoms in the body of ρ are assigned to some of the entries of $\bar{a}'\bar{b}'$ by τ and by τ we know that they are equal to some tuple entries in the corresponding IDB relation.

- The variables occurring in some EDB order atoms in ρ are assigned values in $\bar{a}'\bar{b}'$ which satisfy the EDB order atom by the construction of τ.

- For each variable x occurring in some EDB atom of the form $M(x)$, there are two cases to consider: By the order atoms in τ containing x and some constants, we know from which constant interval I of $\mathcal{A}'$ the assignments to x were. If I is an unchanged interval, the assignment is the same in $\mathcal{A}''$ (for variables occurring in IDBs, this follows from the induction hypothesis). If I is a changed interval,

then the corresponding interval I' contains only elements from M, so for each variable assignment for x from I', the atom $M(x)$ will be satisfied automatically.

Thus, all atoms of ρ are satisfied and $\bar{a}'\bar{b}'$ is a rule instantiation as claimed. □

After deriving a transformation of the underlying structure $\mathcal{A}$ into a simpler form, we discuss how an algorithm solving DATALOG-NONEMPTINESS($\mathcal{A}$) implements these transformations into the datalog program. Similar to Lemma 5.12, we create a modified version of the program such that the tuples in an IDB relation have all the same type. For this task, we only need the structure $\mathcal{A}$ and the finitely many changes from $\mathcal{A}$ to $\mathcal{A}''$ to hardwire them into the algorithm transforming the program. The method for solving DATALOG-NONEMPTINESS introduced in Lemma 6.13 works for an interval sequence order, so we need one further transformation:

Definition 6.22 *Let $\mathcal{A}'' = (A, <, M, c_1, \ldots, c_k)$ be a colored order with empty or uniform infinite constant intervals. Then the* corresponding interval sequence order *$\mathcal{B} = (A, <, I_1, \ldots, I_m, c_1, \ldots, c_k)$ with some $m \leq k+1$ is the interval sequence order created from $\mathcal{A}''$ by the following steps:*

1. *Create the constant intervals as in definition 6.17.*
2. *Remove the empty intervals.*
3. *Label the remaining intervals as $I_1, \ldots, I_m$, where $I_1 < \ldots < I_m$.*

Before discussing how a datalog program is modified according to these transformations, we demonstrate the transformations in a small example:

Example 6.23 ***(For Definition 6.22)***
Consider the colored order $\mathcal{A} = (A, <, M, c_1, c_2)$ with $A = [0,2] \cup \{3,4,5\}$, $M = [0,1] \cup \{4\}$, $c_1 = 0$ and $c_2 = 3$, where $[a,b]$ denotes the interval over the rationals between a and b including endpoints, and (a,b), $(a,b]$ denote intervals excluding both or the lower endpoint respectively.

1. *The constant intervals (see Definition 6.17) are: $I_0 = \emptyset$, $I_1 = (0,3)$, $I_2 = (3,5] = \{4,5\}$*
2. *The transformation in Definition 6.18 adds constants $c_3 = 4$ and $c_4 = 5$ for interval I_2. All other intervals are empty or infinite.*

   Hence $\mathcal{A}' = (A, <, M, c_1, c_2, c_3, c_4)$.
3. *The transformation in Definition 6.20 replaces the interval I_1 (being the only one having a nonempty intersection with M), on interval $(0,1]$. Hence $\mathcal{A}'' = (A', <, M, c_1, c_2, c_3, c_4)$ with $A' = [0,1] \cup \{3,4,5\}$.*
4. *The interval sequence order derived as in Definition 6.22 has the shape: $\mathcal{B} = (A', <, I_1, c_1, c_2, c_3, c_4)$ with $A' = [0,1] \cup \{3,4,5\}$, $I_1 = (0,1]$, $c_1 = 0$, $c_2 = 3$, $c_3 = 4$ and $c_4 = 5$.*

We will now show how to convert the program Π over $\mathcal{A}$ to an interval type disjoint program Π' over the interval sequence order $\mathcal{B}$. After the formal statement, an example visualizes the steps for a program on a simplified version of the structure considered in Example 6.23.

Lemma 6.24 *Let $\mathcal{A}''$ be a colored order with empty or uniform infinite constant intervals and let $\mathcal{B} = (A, <, I_1, \ldots, I_m, c_1, \ldots, c_k)$ be the corresponding interval sequence order. Let Π be a datalog program over $\mathcal{A}''$.*

Then there exists an interval order type disjoint program Π' over $\mathcal{B}$ with the following properties:

1. *For every IDB P of Π there are a number n_P and IDBs $P_1, \ldots, P_{n_P}$ of Π' with pairwise distinct interval order types, such that*

$$P_\infty^{\Pi,\mathcal{A}''} = \bigcup_{j=1}^{n_P} (P_j)_\infty^{\Pi',\mathcal{B}}.$$

2. *$n'_I \in O((k+m)^{m_L} \cdot n_I \cdot 3^{m_L^2})$, $n'_R \in O((k+m)^{m_R} \cdot 3^{m_R^2 \cdot (m_I+1)} \cdot n_R)$, $m'_R = m_R$ and $m'_L = m_L$, $m_I = m'_I$, where n'_I, m'_R, m'_L, m'_I are the parameters of Π'.*

3. *This program can be computed in time exponential in $|\Pi|$.*

Proof: We describe an algorithm which is based on the constants and intervals in $\mathcal{A}''$ and $\mathcal{B}$ and which has access to the information about the membership in M of the constants in $\mathcal{B}$ and the occurring memberships of elements of each interval in $\mathcal{B}$. Altogether, this is a finite set of information which is hardwired into the algorithm. As input, the program Π and the IDB symbol P are given, as output Π' and $P_1, \ldots, P_{n_P}$ are produced. The algorithm runs in three phases: The first creates copies of the rules with complete interval order types, the second ensures that all monadic memberships are correctly transferred to the rules of Π' and the third renames the IDBs accordingly. Note that we do not check the consistency of the EDB order atoms with the interval order type introduced for each rule.[1]

1. For each rule ρ of Π:

   Let $x_1, \ldots, x_\ell$ be the variables of ρ. Let $C = \{I_1, \ldots, I_m, c_1, \ldots, c_k\}$. For each $\bar{\xi} \in C^\ell$ and each complete order type $\gamma \in \Gamma_1(x_1, \ldots, x_\ell)$:

   If for some i, j: $\{(x_i < x_j), (x_i = x_j)\} \cap \gamma \neq \emptyset$ and $\xi_j < \xi_i$, drop this combination $\gamma, \bar{\xi}$ and continue with the next.

   Otherwise, we add a rule $\rho_{\bar{\xi},\gamma}$ to Π' created from ρ with the following modifications:

[1] This consistency check is carried out in the calculation of the initialization sequence in Lemma 6.13.

   (a) If $\xi_i = I_j$ for some i, j, add $I_j(x_i)$ to $\rho_{\bar{\xi},\gamma}$.
   (b) If $(x_i < x_j) \in \gamma$ for some i, j and $\xi_i = \xi_j$, add $x_i < x_j$ to $\rho_{\bar{\xi},\gamma}$.
   (c) If $\xi_i = c_j$ for some i, j, replace all occurrences of x_i by c_j in $\rho_{\bar{\xi},\gamma}$.
   (d) If $(x_i = x_j) \in \gamma$ for some i, j, replace all occurrences of x_i by x_j in $\rho_{\bar{\xi},\gamma}$ (or by the corresponding variable or constant, if x_j has been replaced earlier).

2. For each rule ρ of Π':
   (a) If there is an atom $M(c_i)$ in ρ for some i, but $c_i \notin M$, then delete rule ρ.
   (b) If ρ contains atoms $M(x_i)$ and $I_j(x_i)$ for some i, j, but $I_j \cap M = \emptyset$, delete ρ.
   (c) Remove all atoms of the form $M(x)$ or $M(c_i)$ from ρ.

3. The complete interval order type of the variables of each rule implies a complete interval order type of the head variables, which we will call head IDB interval order type.
   (a) Let for each IDB P $\gamma_1^P, \ldots, \gamma_p^P$ be the head interval order types of rules with head IDB P.
   (b) For each IDB P and each rule with head IDB P and interval order type γ_i^P $(i = 1, \ldots, p)$, replace the head IDB by a new IDB P_i.
   (c) For each IDB P and each rule with an occurrence of P, determine for each occurrence the interval order type of the variables in this occurrence, determine the corresponding interval order type γ_i^P and then replace this occurrence of P by P_i. If no corresponding γ_i^P can be found, delete the rule.

The program created by this algorithm satisfies the first property of the claim: In the first phase, each rule and hence each IDB is sliced into all interval order types, that may occur in this rule. The tuples in the IDB relations computed by Π and by the program created in the first phase coincide:

- Assume that a tuple $\bar{a}$ is added using a rule of Π. Since the structure $\mathcal{B}$ has the same universe as $\mathcal{A}$ and is partitioned into intervals and constants, the entries of $\bar{a}$ will all be contained in intervals or be equal to some constants. In our algorithm, all possible combinations of intervals and constants are considered, so the constants and intervals corresponding to $\bar{a}$ occur in some rule, hence this rule of the modified program can be applied to add tuple $\bar{a}$.

- Assume that a tuple $\bar{a}$ is added using a rule ρ' of the modified program. This rule was created from a rule ρ of Π which may additionally include some atoms of the form $M(x)$. When ρ' was created from ρ, in phase 2 the variable x was constrained to an interval I (or constant) whose elements are all from M. So each atom $M(x)$ in rule ρ will be satisfied by $\bar{a}$ and since the other atoms of ρ are a subset of the atoms of ρ', ρ can be used to add $\bar{a}$ to the relation.

In the third phase of the algorithm, each IDB is sliced into sets of tuples with the same interval order type, by definition. The rules deleted in this step correspond to inconsistent types and hence nonexistent tuples. Thus, the first claim holds.

The bounds of the parameters of the program Π' are rough upper bounds by counting the number of possibilities in the worst case. For n'_I note that each IDB is sliced into different interval order types for its at most m_L variables. In each type, each variable is contained in an interval or equal to a constant, adding up to $(k+m)^{m_L}$ possibilities. If some variables are in the same interval, their order type has to be considered, for at most m_L variables less than $3^{m_L^2}$ different possibilities, which is the last factor. The bound for m_R is derived analogously.

For the running time, we have to consider the loops involved in the sketched algorithm. The loops are clearly within the exponential bound, since the outer loop iterates over the rules and the inner loop iterates over some types which are at most exponentially many (as in the preceding paragraph). □

Example 6.25 *In this example, we take a simplified version of the structures from Example 6.23:* $\mathcal{A} = ((0,2] \cup \{3,4\}, <, M = (0,1] \cup \{4\}, c_2 = 3)$ *and* $\mathcal{B} = ((0,1] \cup \{3,4\}, <, I_1 = (0,1], c_2 = 3, c_3 = 4)$.

Let Π *be the following datalog program over* $\mathcal{A}$*:*

$$\begin{aligned} \rho_1: &\quad P(x,y) \leftarrow M(y),\ x < y. \\ \rho_2: &\quad Q(x) \leftarrow P(x, c_2). \end{aligned}$$

We first sketch the phases 1 and 2 of the algorithm, for both rules. For the first rule ρ_1*, we only consider the order type* $\gamma = \{(x < y)\}$*. For the other types, the algorithm works similar (but produces inconsistent rules in these cases).*

ξ_1	ξ_2	*Phase 1*	*Phase 2*
I_1	I_1	$P(x,y) \leftarrow M(y),\ x < y,\ I_1(x),\ I_1(y).$	
I_1	c_2	$P(x,c_2) \leftarrow M(c_2),\ x < c_2,\ I_1(x).$	
I_1	c_3	$P(x,c_3) \leftarrow M(c_3),\ x < c_3,\ I_1(x).$	$M(c_3) \Rightarrow$ *rule deleted*
c_2	I_1	$I_1 < c_2 \Rightarrow$ *rule deleted*	
c_2	c_2	$c_2 = c_2 \Rightarrow$ *rule deleted*	
c_2	c_3	$P(c_2,c_3) \leftarrow M(c_3),\ c_2 < c_3.$	$M(c_3) \Rightarrow$ *rule deleted*
c_3	I_1,c_2, c_3	*all deleted*	
I_1		$Q(x) \leftarrow P(x,c_2),\ I_1(x).$	
c_2		$Q(c_2) \leftarrow P(c_2,c_2).$	
c_3		$Q(c_3) \leftarrow P(c_3,c_2).$	

In the third phase the following program is created from these rules, where we omit the inconsistent rules again:

$$P_1(x,y) \leftarrow x < y,\ I_1(x),\ I_1(y).$$
$$P_2(x,c_2) \leftarrow x < c_2,\ I_1(x).$$
$$Q_1(x) \leftarrow P_2(x,x_2),\ I_1(x).$$

With this lemma we are able to show the main claim of this section:

Theorem 6.26 *On linear orders $\mathcal{A} = (A, <, M, c_1, \ldots, c_r)$ with finitely many constants (which may be used by the datalog programs) and one monadic relation M DATALOG-NONEMPTINESS($\mathcal{A}$) can be decided in exponential time.*

Proof: This is an implementation of our scheme from Remark 6.14.

Following Definitions 6.18 and 6.20 there is a colored order with empty or uniform infinite constant intervals $\mathcal{A}''$ to $\mathcal{A}$, such that for the program Π the nonemptiness of each IDB is equivalent on $\mathcal{A}$ and $\mathcal{A}''$ by Lemmas 6.19 and 6.21. Definition 6.22 shows how to derive an interval sequence order $\mathcal{B}$ to $\mathcal{A}''$ and Lemma 6.24 shows how to derive an interval order type disjoint program Π' on $\mathcal{B}$ from Π in exponential time, such that the nonemptiness of an IDB P of Π on $\mathcal{A}''$ is equivalent to the nonemptiness of a finite set of IDBs of Π' on $\mathcal{B}$.

Using Lemma 6.13, we can compute the nonemptiness of all IDBs of Π' on $\mathcal{B}$ in time polynomial in $|\Pi'|$, which is exponential in $|\Pi|$ by Lemma 6.24. Hence to check if IDB P of Π is nonempty on $\mathcal{A}$, we only have to check if one of the at most exponentially many IDBs $P_1, \ldots, P_s$ of Π' is nonempty and have solved the problem. □

We can use this method for a scenario introduced in Section 5.6, datalog on orders with constants, but without colors. In this case, we just ignore the monadic relation M, e.g. by letting $M = \emptyset$ and solve DATALOG-NONEMPTINESS with this monadic relation, leading to:

Corollary 6.27 *On linear orders $\mathcal{A} = (A, <, c_1, \ldots, c_r)$ with finitely many constants (which may be used by the datalog programs) DATALOG-NONEMPTINESS($\mathcal{A}$) can be decided in exponential time.*

6.3 Colored Orders With Constants

In this section we will show how to transfer the methods introduced for colored orders with one monadic relation to colored orders with two or more monadic relations. As motivation for a sophisticated type concept which does not simply count the number of elements in each monadic relation, we had a look at a structure with two monadic relations in Example 6.3. In this example the elements contained in the monadic EDB relations M_1 and M_2 occur in an alternating way making it impossible to define two disjoint intervals that contain elements from one monadic EDB only. To be

able to transfer the information of the monadic memberships (i.e. the memberships of elements in monadic EDB relations) to an interval sequence order, we have split the structure into smaller intervals. After this process each interval will contain elements with a homogenous monadic membership only. For a finite structure like in Example 6.3, we simply collect consecutive elements with the same membership into an interval:

Example 6.28 *Let $\mathcal{A} = (A, <, M_1, M_2)$ be the structure from Example 6.3, i.e. $A = \{1, 2, 3, 4, 5, 6\}$, $<^{\mathcal{A}}$ being the usual order, $M_1 = \{1, 3, 6\}$ and $M_2 = \{2, 4, 5\}$.*

A partition of A into disjoint intervals where each interval contains only elements with one monadic membership is the following set of intervals:

	I_1	I_2	I_3	I_4	I_5
Interval	$[1, 1]$	$[2, 2]$	$[3, 3]$	$[4, 5]$	$[6, 6]$
Monadic Membership	M_1	M_2	M_1	M_2	M_1

For infinite structures, we may not create such a finite partition any more, and on infinite orders containing dense parts, extra care has to be taken to define intervals consisting of elements with different monadic EDB memberships. It may also be the case that the interval bounds needed are not part of the structure, as the following example shows.

Example 6.29 *Let $\mathcal{A} = (\mathbb{Q}, <, M_1, M_2)$ be the ordered rational numbers and let $M_1 = \left\{x \in \mathbb{Q} \;\middle|\; x < \sqrt{2}\right\}$ and $M_2 = \left\{x \in \mathbb{Q} \;\middle|\; \sqrt{2} < x\right\}$. Then we can clearly split this structure into the intervals $I_1 = M_1$ and $I_2 = M_2$, which are disjoint and ordered $I_1 < I_2$. However, the upper bound of the open interval I_1 and the lower bound of I_2 is $\sqrt{2}$, which is not in $\mathcal{A}$.*

We will now prove a lemma in which this effect has to be considered and in the proof it may be necessary to choose intervals with bounds not in the structure, but from some completion. However, when we employ this result later on to convert the datalog programs, there is no need for using the interval bounds explicitly. Rather than using constants as interval bounds in datalog programs, intervals will just be considered as special sets (or monadic relations) with the properties as defined in Definition 6.4.

6.3.1 A Partition Lemma for Infinite Linear Orders

Before we concentrate on datalog programs on colored orders with more than one monadic relation, we will show a lemma, which helps to classify the monadic memberships of elements in intervals. On colored orders with more than one monadic relation, each element may be in one or more of the monadic relations, leading to 2^m different memberships in the case of m monadic relations. To keep things clearer, we present a more abstract version, where the monadic memberships are replaced by a finite partially ordered set Σ, and the memberships of the elements are replaced by a function φ.

Lemma 6.30 *Let $k > 1$ and $\Sigma = \{\sigma_0, \ldots, \sigma_{k-1}\}$ be a finite set with a partial order $\leq^\Sigma$ on Σ, S be an infinite linearly ordered set and $\varphi : S \to \Sigma$ a mapping from S to the set Σ satisfying*

$$\bigcup_{s \in S} \{\varphi(s)\} = \Sigma \tag{6.4}$$

Then exactly one of the following cases holds:

P1 : There is an infinite sequence $s_0, s_1, \ldots, s_{k-1}, \ldots$ satisfying for all $j \geq 0$

$$\sigma_{j \mod k} \leq^\Sigma \varphi(s_j) \tag{6.5}$$

P2 : There exist an $\ell \in \mathbb{N}$ and a partition of S into (disjoint) intervals $I_0 < I_1 < \cdots < I_\ell$, such that for each interval I_j the condition

$$\psi(I_j) := |\left\{\sigma_i \;\middle|\; \exists x \in I_j \text{ with } \sigma_i \leq^\Sigma \varphi(x)\right\}| < k$$

holds.

Informally, condition *P1* states that there is an infinite sequence whose elements dominate each element of Σ infinitely often, in some cycling fashion. By "x dominates σ" we will refer to $\sigma \leq^\Sigma \phi(x)$ and use this phrase analogously for sets.

Proof: If there exists a partition into intervals satisfying property *P2*, it is obvious that there may not be a sequence satisfying property *P1*. For the other direction of the proof we assume that there is no sequence satisfying property *P1* and we will create a partition into intervals with *P2*.

Since there is no sequence with property *P1*, there is a maximal m, such that there exists a finite sequence $\tilde{s} = (s_1, \ldots, s_m)$ in S such that for $1 \leq j \leq m$, the condition (6.5) is satisfied. We define a partition into intervals:

$$\begin{aligned} I_0 &= \{x \in S \mid x \leq s_0\} \\ I_j &= \{x \in S \mid s_{j-1} < x \leq s_j\} \quad \text{for all } j \in \{1, \ldots, m\} \\ I_m &= \{x \in S \mid s_m < x\} \end{aligned}$$

If this partition already has the property *P2*, we have found a suitable partition. Otherwise, there is at least one interval I_j with $\psi(I_j) = k$ and for each such interval we show that it can be partitioned into a partition satisfying *P2*. Since there are only finitely many intervals $I_0, \ldots, I_m$, we may combine the intervals I_j with $\psi(I_j) < k$ and the partitions of the other intervals into a partition satisfying *P2* as claimed.

Now let I_j be an interval with $\psi(I_j) = k$. We will create a partition of I_j with property *P2* based on the fact that $\tilde{s}$ has maximal length m.

Note that I_j is bounded from below by s_{j-1} and from above by s_j with $\sigma_{j-1 \mod k} \leq^\Sigma \varphi(s_{j-1})$ and $\sigma_{j \mod k} \leq^\Sigma \varphi(s_j)$; for the interval I_0 we may w.o.l.g. assume $s_{-1} = -\infty$ with $\sigma_{k-1} \leq^\Sigma \varphi(s_{-1})$ as lower bound and for I_m, $s_{m+1} = \infty$ with $\sigma_{m+1 \mod k} \leq^\Sigma \varphi(s_{m+1})$ as upper bound.

To create the partition into intervals, we create a sequence $s_{j-1} < \xi_1 < \xi_2 < \cdots < \xi_\nu < s_j$ with maximal ν satisfying for all i:

- $\sigma_{j+i-1 \mod k} \leq^{\Sigma} \varphi(\xi_i)$ and
- the interval $(s_{j-1}, \xi_i]$ can be partitioned into finitely many intervals with property *P2*.

The maximality of m yields $\nu < k$, since otherwise we would have a longer sequence $\tilde{s}'$ (of length at least $m + \nu$) satisfying *P1*. On the other hand, the maximality of ν has the consequence that the interval (ξ_ν, s_j) does not contain an element dominating $\sigma_{j+\nu \mod k}$, and hence $\psi(\,(\xi_\nu, s_j)\,) < k$. Then the intervals partitioning $(s_{j-1}, \xi_\nu]$ together with this interval form a partition of I_j with property *P2*.

It only remains to show how the elements $\xi_1 < \xi_2 < \cdots < \xi_\nu$ are chosen. Iterating over $i = 1, 2, \ldots, k-1$, we check if the interval (ξ_{i-1}, s_j) contains an element dominating $\sigma_{j+i-1 \mod k}$, where $\xi_0 = s_{j-1}$. If no such element can be found, we have created the whole sequence of ξ_i's, otherwise we choose ξ_i:

Let J_i be the biggest interval of elements that are all smaller than all elements dominating $\sigma_{j+i-1 \mod k}$:

$$J_i = \Big\{ x \in I_j \;\Big|\; y \in I_j \wedge \sigma_{j+i-1 \mod k} \leq^{\Sigma} \varphi(y) \;\Rightarrow\; x < y \Big\}$$

Since no element in J_i dominates $\sigma_{j+1-1 \mod k}$, $\psi(J_i) < k$. Then the interval $J' = \{ x \mid J_i < x < s_j \}$ is nonempty and contains some x dominating $\sigma_{j+i-1 \mod k}$. Let J_x be the interval between J_i and x:

$$J_x = \{ y \in J' \mid J_i < y < x \}$$

We choose $\xi_i \in J'$ dominating $\sigma_{j+i-1 \mod k}$ and satisfying $\psi(J_x) < k$, which is always possible:

> Assume that for all $x \in J'$ with $\sigma_{j+i-1 \mod k} \leq^{\Sigma} \varphi(x)$, $\psi(J_x) = k$. We choose such an x as x_{k-1}.
>
> By $\psi(J_{x_{k-1}}) = k$ we find an element y_{k-1} in $J_{x_{k-1}}$ with $\sigma_{j+k-1 \mod k} \leq^{\Sigma} \varphi(y_{k-1})$. Since $y_{k-1} \in J'$, there is an $x_{k-2} \in J'$ with $\sigma_{j+i-1 \mod k} \leq^{\Sigma} \varphi(x_{k-2})$ and $x_{k-2} < y_{k-1}$. By our assumption $\psi(J_{x_{k-2}}) = k$ and we may choose an $y_{k-2} \in J_{x_{k-2}}$ with $\sigma_{j+k-2 \mod k} \leq^{\Sigma} \varphi(y_{k-2})$. Continuing until y_0, we extend the finite sequence $\tilde{s}$ by $y_0 < \cdots < y_{k-1}$:
>
> $$s_1 < \ldots < s_{j-1} < y_0 < \cdots < y_{k-1} < s_j < \cdots < s_m$$
>
> By construction, this sequence will satisfy the condition (6.5), a contradiction to the maximality of m. Hence we may choose $x \in J'$ such that $\psi(J_x) < k$ and x dominates $\sigma_{j+i-1 \mod k}$.

□

6.3.2 Monadic Memberships

To apply the lemma to colored orders, we define a description of the monadic relations by a set of monadic memberships. Each of the elements in a colored order $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ may be member of each of the monadic relations $M_1, \ldots, M_m$, so we have up to 2^m different types of memberships in monadic relations present in A. We denote the set of memberships by $\mu^* = \{\mu_0, \ldots, \mu_{2^m-1}\}$, where semantics behind this notation is understood as:

Definition 6.31 ***(Monadic Membership)***
An element $a \in A$ has membership μ_i if μ_i is the set of all monadic relation symbols M_j with $a \in M_j$.

A monadic membership mapping is a mapping $\varphi : A \to \mu^*$, returning the membership for each element of A.

The monadic memberships of a set $S \subseteq A$ is the set $\varphi(S)$.

Note that the membership set μ^* is partially ordered by set inclusion. We extend this partial order to vectors of memberships:

Definition 6.32 *Let $k \in \mathbb{N}$ and let $\bar{\mu}, \bar{\nu} \in (\mu^*)^k$. Then we define $\preccurlyeq$ as:*

$$\bar{\mu} \preccurlyeq \bar{\nu} \qquad \Longleftrightarrow \qquad \text{for all } i = 1, \ldots, k: \quad \mu_i \subseteq \nu_i$$

We define the set membership function φ on $(\mu^*)^k$ as:

$$\varphi(\mu) \;:=\; (\varphi(\mu_1), \ldots, \varphi(\mu_k))$$

The set μ^* and the restriction of φ to intervals will be used for applying Lemma 6.30. For intervals which cannot be partitioned into smaller intervals containing less elements from μ^*, Lemma 6.30 guarantees the existence of an infinite sequence with cycling membership constraints.

6.3.3 Structure Transformations

As first step in the implementation of the plan from Remark 6.14 we derive a finite set of transformations of the colored order $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ into an interval sequence order $\mathcal{B} = (B, <, I_1, \ldots, I_s, c_1, \ldots, c_t)$. We use the structure $\mathcal{A}$ and this finite set of transformations to create an algorithm to derive an interval order type disjoint program Π' over $\mathcal{B}$ for each program Π over $\mathcal{A}$ with which we can solve DATALOG-NONEMPTINESS for Π.

The first stage of our transformation will be the inductive application of Lemma 6.30 to transform the constant intervals of $\mathcal{A}$ into intervals with a restricted appearance of monadic memberships, while the second stage is an analogon to Definition 6.20, where all finite intervals are replaced by constants.

Definition 6.33 *Let $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ be a colored order and let $\varphi : A \to \mu^*$ be the mapping denoting the monadic memberships of the elements in A, where μ^* denotes all occurring monadic memberships in A. $\mathcal{A}^i = (A_i, <, I_1, \ldots, I_s, c'_1, \ldots, c'_t)$ is an **interval sequence order to** $\mathcal{A}$, if $\mathcal{A}^i$ is an interval sequence order derived from $\mathcal{A}$ by applying the following steps:*

1. *Compute the list $\mathcal{L}$ of constant intervals $I_0, \ldots, I_k$ of $\mathcal{A}$; copy the constants into a list $\mathcal{C}$.*
2. *If an* infinite *interval I from $\mathcal{L}$ has a partition with property* P2 *when Lemma 6.30 is applied to $S = I$ and $\Sigma = \varphi(I)$, then delete I from the list $\mathcal{L}$ and add the finitely many intervals of the partition of I. Iterate step 2 for this new list $\mathcal{L}$.*
3. *Let $A^i = A$. For each infinite interval I from $\mathcal{L}$:*

   By Lemma 6.30 with $S = I$ and $\Sigma = \varphi(I)$, I contains a sequence $s_0, s_1, \ldots$ with property P1*. Delete the elements of I from B and add the elements $s_0, s_1, \ldots$.*
4. *For each finite interval I from $\mathcal{L}$, add a new constant to $\mathcal{C}$ for each element in I and remove I from $\mathcal{L}$.*
5. *Order $\mathcal{C}$ and $\mathcal{L}$ according to $<$.*
6. *$\mathcal{A}^i = (A^i, <, I_1, \ldots, I_s, c'_1, \ldots, c'_t)$, where $I_1, \ldots, I_s$ are the intervals from $\mathcal{L}$ and $c'_1, \ldots, c'_t$ are the constants from $\mathcal{C}$.*

*The **colored interval sequence order** $\mathcal{A}^c$ to $\mathcal{A}$ is the structure order $\mathcal{A}^c = (A_i, <, I_1, \ldots, I_s, M_1^c, \ldots, M_m^c, c'_1, \ldots, c'_t)$, where for each j, $M_j^c = M_j \cap A^i$.*

Having a look at these transformation steps it is unclear if there are only finitely many steps involved, especially the transformations carried out in Step 2 could be infinitely many. To see that they are only finitely many, we use an inductive argument:

At the beginning, the list $\mathcal{L}$ contains only finitely many intervals. In each execution of Step 2 an interval I with $\psi(I) = \ell$, for some number ℓ of different monadic memberships, see Lemma 6.30, is considered. Each interval J created in the partition of I has strictly less monadic memberships, $\psi(J) < \psi(I)$.

Since each of the partitions contains only finitely many smaller intervals and there are only finitely many different monadic memberships, Step 2 can only be executed a finite number of times.

The resulting structure $\mathcal{A}^i$ satisfies all properties of an interval sequence order and moreover its intervals have the following properties:

Lemma 6.34 *Let $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ be a colored order, let $\varphi : A \to \mu^*$ be the mapping denoting the monadic memberships of the elements in A, where μ^* denotes all monadic memberships occurring in A and let $\mathcal{A}^i$ be the interval sequence order to $\mathcal{A}$ as above. Then each interval I_j with $1 \leq j \leq s$ satisfies the following properties:*

- $|I| = \infty$
- *Either* $|\varphi(I)| = 1$ *or* I *contains a sequence of cycling monadic memberships, i.e. applying Lemma 6.30 to* $S = I$ *and* $\Sigma = \varphi(I)$*, there is a sequence* $s_0, s_1, \ldots$ *in* I *with property* P1*.*

Proof: The first claim follows immediately from Step 4, since each finite interval is removed. The second claim follows from Steps 2: After all iterations of Step 2, there are no intervals with a partition satisfying property *P2* of Lemma 6.30. So each remaining interval I has to satisfy either $|\varphi(I)| = 1$, in which case all elements have the same monadic membership, or for $|\varphi(I)| > 1$ there has to be a sequence $s_0, s_1, \ldots$ with property *P1* in I. The steps following Step 2 do not change this property, since Step 3 keeps the sequence and Step 4 affects only finite intervals. □

Note that $\mathcal{A}^c$ is simply $\mathcal{A}^i$ with some additional monadic relations. Before we show how to transform each program Π over $\mathcal{A}$ to an interval disjoint version over $\mathcal{A}^i$, we will show that for each program Π on $\mathcal{A}$ the nonemptiness of each relation remains unchanged when evaluating Π on $\mathcal{A}^c$. The following lemma is an adaptation of Lemma 6.21 to this context.

Lemma 6.35 *Let* $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ *be a colored order, let* $\varphi : A \to \mu^*$ *be the mapping denoting the monadic memberships of the elements in* A*, where* μ^* *denotes all monadic memberships occurring in* A*.*

Let $\mathcal{A}^c = (A_i, <, I_1, \ldots, I_s, M_1^c, \ldots, M_m^c, c_1', \ldots, c_t')$ *be the colored interval sequence order to* $\mathcal{A}$*.*

Let Π *be a program over* $\mathcal{A}$ *and let* Π' *be a copy of* Π*, in which each constant symbol from* $\mathcal{A}$ *is replaced by the corresponding constant symbol from* $\mathcal{A}^c$ *and each monadic IDB from* $\mathcal{A}$ *by the corresponding monadic IDB from* $\mathcal{A}^c$*. Let* P *be an IDB of* Π *(and the corresponding IDB of* Π'*). Then it holds that*

$$P_\infty^{\Pi,\mathcal{A}} \neq \emptyset \quad \Longleftrightarrow \quad P_\infty^{\Pi',\mathcal{A}^c} \neq \emptyset$$

Proof: Since $A^i \subseteq A$ and $\mathcal{A}^c$ contains all monadic relations and constants of $\mathcal{A}$ and Π only uses these symbols (not the additional constants or the intervals), the implication from $\mathcal{A}^c$ to $\mathcal{A}$ is immediate.

For the other implication, we will prove the following claim by induction on the stages of the evaluation of Π (and Π'):

> Let ρ_i be the rule applied in stage i. Let $(\rho_0, \ldots, \rho_{i-1})$ be the rules applied in the previous stages and let $\xi \in \mathbb{N}$. Let $\mu \in (\mu^*)^\xi$ be a vector of monadic memberships and let $\tau \in \Gamma_1(x_1, \ldots, x_\xi)$ be an order type.
>
> Let $\bar{c}$ denote the tuple consisting of the constants of $\mathcal{A}$ (which is equal to the tuple consisting of the corresponding constants of $\mathcal{A}^c$).
>
> Then **C1** implies **C2**, which are as follows:

C1: For all j with $0 \leq j \leq i$, there is an instantiation $\bar{a}^{(j)}\bar{b}^{(j)}$ of rule ρ_j of Π with elements of A such that

- the instantiation $\bar{a}^{(i)}\bar{b}^{(i)}$ of rule ρ_i only needs the elements of these preceeding instantiations in the IDB relations, and
- there is a tuple e of elements from A such that the concatenation

$$\bar{g} := \bar{c}\bar{a}^{(0)}\bar{b}^{(0)} \cdots \bar{a}^{(i)}\bar{b}^{(i)}\bar{e}$$

of tuples satisfies τ and $\mu \preccurlyeq \varphi(\bar{g})$.

C2: For all j with $0 \leq j \leq i$, there is an instantiation $\bar{a'}^{(j)}\bar{b'}^{(j)}$ of rule ρ_j of Π' with elements of A^i such that

- the instantiation $\bar{a'}^{(i)}\bar{b'}^{(i)}$ of rule ρ_i only needs the elements of these preceeding instantiations in the IDB relations, and
- there is a tuple e' of elements from A^i such that the concatenation

$$\bar{g'} := \bar{c}\bar{a'}^{(0)}\bar{b'}^{(0)} \cdots \bar{a'}^{(i)}\bar{b'}^{(i)}\bar{e'}$$

of tuples satisfies τ and $\mu \preccurlyeq \varphi(\bar{g'})$.

Induction base: The rule ρ does not contain IDB atoms in the body. The EDB atoms are order atoms and monadic membership atoms.

We extend τ to τ' by adding all order atoms of ρ. If in ρ there are multiple occurrences of a variable in the head IDB atom, then the corresponding tuple entries of the instantiation have to be equal and we add a corresponding equality atom to τ'. For each constant in the head IDB atom of ρ, we add an equality atom with the tuple entry and the constant.

Let μ' be a vector of monadic memberships, which is created from μ by componentwise union of the monadic memberships induced by the monadic atoms of rule ρ.

Then $\bar{g}$ satisfies τ' and $\mu \preccurlyeq \mu' \preccurlyeq \varphi(\bar{g})$. We will now create the tuple $\bar{g}'$ in $\mathcal{A}^c$ with $\mu' \preccurlyeq \varphi(\bar{g}')$ and satisfying τ'.

Each entry of $\bar{g}$ that is *not* from one of the intervals replaced by a sequence in step 3 of Definition 6.33 is simply copied to the same position in $\bar{g}'$. These elements are from parts of the universe A remaining unchanged and retain their monadic memberships and their order type in $\mathcal{A}^c$.

For the other entries of $\bar{g}$, we consider all elements in one interval modified in step 3. Since the intervals are linearly ordered and we carry out the replacements only within one interval at a time, the order type with the elements in other intervals or constants will not change. Let I be an interval which is replaced by a sequence $s = (s_1, s_2, \dots)$ in step 3 of Definition 6.33 and which contains some elements of $\bar{g}$.

We introduce two abbreviations for an easier notation: We collect the entries of $\bar{g}$ in I in a tuple $\bar{h}$, which is ordered ascending. In $\bar{g}'$ there is a corresponding tuple $\bar{h}'$ and when we assign some values to entries of $\bar{h}'$, it will be an abbreviation for setting

the corresponding entries of $\bar{g}$. For an entry x of $\bar{h}$, let $\mu'(x)$ denote the element of μ' at the same position as x in $\bar{g}$.

h_1 is the smallest entry of $\bar{h}$ and there may be some entries $h_2, \ldots, h_j$ with values equal to h_1. If we collect all values $\mu'(h_\ell)$ for $\ell = 1, \ldots, j$ in a monadic membership ν (by component-wise union) then $\varphi(h_\ell) \subseteq \nu$ for all ℓ. So I contains an element dominated by the monadic membership ν and by the construction of s there is a smallest index κ with $\nu \subseteq \varphi(s_\kappa)$, hence for all ℓ: $\varphi(h_\ell) \subseteq \varphi(s_\kappa)$. We choose $h'_i = s_\kappa$ for each h_ℓ with $1 \leq \ell \leq j$. For the element h_{j+1} we analogously determine a κ' such that the conditions of the monadic memberships are satisfied, but with $\kappa' > \kappa$. Since s is infinite and contains all memberships present in I infinitely often in a cycling manner, we will find such a κ' with a sequence element with monadic membership dominating $\mu'(h_{j+1})$ (possibly combined with memberships of other entries with values equal to h_{j+1}).

After determining replacement elements $\bar{h}'$ for the whole tuple $\bar{h}$, this tuple satisfies all order types that $\bar{h}$ satisfies, since the entries appear in the same order. And it satisfies $\mu'(\bar{h}) \preccurlyeq \mu'(\bar{h}')$.

After finding replacement elements for all entries in intervals that were replaced by a sequence, the tuple $\bar{g}'$ created satisfies τ' and $\mu'(\bar{g}) \preccurlyeq \mu'(\bar{g}')$.

It remains to show that the part $\bar{a'}^{(0)}\bar{b'}^{(0)}$ of this tuple is an instantiation of ρ (as rule of Π'). We have created τ' to contain all order atoms of ρ and atoms taking care of the equality of tuple entries for variables occurring more than once in ρ_i and tuple entries being equal to constants. We have created μ' so that for a tuple satisfying μ' the part corresponding to the variables of ρ satisfy all monadic atoms. Hence all atoms of ρ are satisfied by $\bar{a'}^{(0)}\bar{b'}^{(0)}$ and therefore it is an instantiation of ρ.

Induction step: The rule ρ contains IDB atoms in the body and may also contain EDB order atoms and monadic EDB atoms in the body. As in the induction base, we extend τ to τ' and μ to μ' to contain all the monadic atoms of ρ and ensure that entries of $\bar{g}$ assigned to the same variable are equal. Additionally, we add equality atoms for all variables of ρ occurring in IDB atoms of the body of ρ: If a variable x is assigned an entry a of $\bar{a}^{(i)}\bar{b}^{(i)}$ in the instantiation of ρ and x also occurs in an IDB atom Q, the tuple used from the relation Q was added in some previous instantiation $\bar{a}^{(j)}\bar{b}^{(j)}$ with $j < i$ and this instantiation contains an entry b corresponding to x in ρ. We add an atom $a = b$ to τ' (where a and b denote the entries of $\bar{g}$). These atoms of τ' ensure that each satisfying tuple $\bar{g}$ contains a consistent variable assignment of ρ.

Then $\bar{g}$ satisfies τ' and $\mu \preccurlyeq \mu' \preccurlyeq \varphi(\bar{g})$. We will now create the tuple $\bar{g}'$ with $\mu' \preccurlyeq \varphi(\bar{g}')$ and satisfying τ'.

We apply the induction hypothesis for $i - 1$ on the tuple $\bar{g}$. In this case, the instantiation $\bar{a}^{(i)}\bar{b}^{(i)}$ of ρ in the i-th stage is considered as part of the tuple $\bar{e}$: The induction hypothesis is applied to:

$$\bar{g} := \bar{c}\bar{a}^{(0)}\bar{b}^{(0)} \cdots \bar{a}^{(i-1)}\bar{b}^{(i-1)}\bar{e^*}$$

with $\bar{e^*} = \bar{a}^{(i)}\bar{b}^{(i)}\bar{e}$.

By the induction hypothesis we have a tuple

$$\bar{g'} := \bar{c}\bar{a'}^{(0)}\bar{b'}^{(0)} \cdots \bar{a'}^{(i-1)}\bar{b'}^{(i-1)}(\bar{e^*})'$$

with the properties from **C2**. Using the arity of $\bar{a}^{(i)}\bar{b}^{(i)}$ we can split $(\bar{e^*})'$ into:

$$(\bar{e^*})' = \bar{a'}^{(i)}\bar{b'}^{(i)}\bar{e'}$$

Then the tuple $\bar{g'}$ satisfies τ' and hence also $\tau \subseteq \tau'$ and $\mu \preccurlyeq \mu' \preccurlyeq \varphi(\bar{g'})$. We only have to show that $\bar{a'}^{(i)}\bar{b'}^{(i)}$ is an instantiation of ρ (as rule of Π').

By the construction of τ', $\bar{a'}^{(i)}\bar{b'}^{(i)}$ is a consistent variable assignment of ρ and it satisfies all order EDBs of ρ. By construction of μ', all monadic atoms of ρ are satisfied as well.

Now consider an IDB atom of ρ. By construction of τ', the entries of $\bar{a'}^{(i)}\bar{b'}^{(i)}$ assigned to variables in this IDB atom are equal to some $\bar{a'}^{(j)}$ with $j < i$. Hence this IDB atom is also satisfied.

Since all atoms of ρ are satisfied, $\bar{a'}^{(i)}\bar{b'}^{(i)}$ is indeed an instantiation of ρ as claimed. □

Although in this proof the inductive argument was using an induction over the number of rule applications, at each stage the elements of all previous stages were considered as well and elements of future stages needed to compute the nonemptiness of an IDB relation: the tuple $\bar{e}$. Using this technique we could ensure that the replacement of some elements in the current stage will allow suitable replacement elements for the rest of the computation.

6.3.4 Program Transformations

Having defined an interval sequence order $\mathcal{A}^i$ to a colored order $\mathcal{A}$, we define a transformation of a datalog program Π over $\mathcal{A}$ to an interval order type disjoint program Π' over $\mathcal{A}^i$. Since we know that the nonemptiness of the relations of a program Π on $\mathcal{A}$ is equivalent to the nonemptiness of the relations on the colored interval sequence order $\mathcal{A}^c$, we describe a transformation of a program Π over $\mathcal{A}^c$ to an interval order type disjoint program Π' over $\mathcal{A}^i$. This transformation will not affect the nonemptiness of IDB relations as we will show in the next lemma. The transformation is a close adaption of the algorithm in Lemma 6.24.

Definition 6.36 *Let $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ be a colored order, let $\varphi : A \to \mu^*$ be the mapping denoting the monadic memberships of the elements in A, where μ^* denotes all occurring monadic memberships in A. Let $\mathcal{A}^i = (A_i, <, I_1, \ldots, I_s, c'_1, \ldots, c'_t)$ and $\mathcal{A}^c$ be as in Definition 6.33. The following algorithm takes as input a datalog program Π over $\mathcal{A}$ and creates a program Π' over $\mathcal{A}^i$. We call the program Π' created by this algorithm the* interval order type disjoint version of Π.

The algorithm runs in three phases:

1. *For each rule ρ of Π:*

   Let $x_1, \ldots, x_\ell$ be the variables of ρ. Let $C = \{I_1, \ldots, I_m, c'_1, \ldots, c'_t\}$. For each $\bar{\xi} \in C^\ell$ and each complete order type $\gamma \in \Gamma_1(x_1, \ldots, x_\ell)$:

   If for some i, j $\{(x_i < x_j), (x_i = x_j)\} \cap \gamma \neq \emptyset$ and $\xi_j < \xi_i$, drop this combination $\gamma, \bar{\xi}$ and continue with the next.

   Otherwise, we add a rule $\rho_{\bar{\xi},\gamma}$ to Π' created from ρ with the following modifications:

   (a) *Replace all constants of $\mathcal{A}$ by the corresponding constants of $\mathcal{A}^i$.*
   (b) *If $\xi_i = I_j$ for some i, j, add $I_j(x_i)$ to $\rho_{\bar{\xi},\gamma}$.*
   (c) *If $(x_i < x_j) \in \gamma$ for some i, j and $\xi_i = \xi_j$, add $x_i < x_j$ to $\rho_{\bar{\xi},\gamma}$.*
   (d) *If $\xi_i = c'_j$ for some i, j, replace all occurrences of x_i by c'_j in $\rho_{\bar{\xi},\gamma}$.*
   (e) *If $(x_i = x_j) \in \gamma$ for some i, j, replace all occurrences of x_i by x_j in $\rho_{\bar{\xi},\gamma}$ (or by the corresponding variable or constant, if x_j has been replaced earlier).*

2. *For each rule ρ of Π':*

   (a) *If there is an atom $M_j(c'_i)$ in ρ for some i, j, but $c'_i \notin M_j$ in $\mathcal{A}^c$, then delete rule ρ.*
   (b) *If ρ contains atoms $M_j(x_i)$ and $I_{j'}(x_i)$ for some i, j, j', but there is no $a \in I_{j'}$ with $M_j \in \varphi(a)$ in $\mathcal{A}^c$, delete ρ.*
   (c) *Remove all atoms of the form $M_j(x)$ or $M_j(c'_i)$ from ρ.*

3. *The complete interval order type of the variables of each rule implies a complete interval order type of the head variables, which we will call head IDB interval order type.*

   (a) *Let for each IDB P $\gamma_1^P, \ldots, \gamma_p^P$ be the head interval order types of rules with head IDB P.*
   (b) *For each IDB P and each rule with head IDB P and interval order type γ_i^P $(i = 1, \ldots, p)$, replace the head IDB by a new IDB P_i.*
   (c) *For each IDB P and each rule with an occurrence of P, determine for each occurrence the interval order type of the variables in this occurrence, determine the corresponding interval order type γ_i^P and then replace this occurrence of P by P_i. If no corresponding γ_i^P can be found, delete the rule.*

Lemma 6.37 *The algorithm in Definition 6.36 computes an interval order type disjoint program Π'. For each IDB P of Π there is a value n_P and a set of IDBs $\{P_1, \ldots, P_{n_P}\}$ of Π' with pairwise different interval order types.*

The running time of this algorithm is in $O(d^{|\Pi|})$ for some suitable constant d depending on $\mathcal{A}$ and also $|\Pi'| \in O(d^{|\Pi|})$.

Proof: The algorithm in Definition 6.36 is based on the monadic memberships and the order of the constants and intervals of $\mathcal{A}^c$.

That the program is in the interval order type form as claimed is immediate from the algorithm. Since the number of intervals and constants is constant, counting the different possibilities in the steps of the algorithms leads to a length of Π' that is at most exponential in $|\Pi|$.

The running time of the algorithm can be analyzed exactly as in Lemma 6.24, with one exception: Phase 2.

Here, monadic memberships have to be considered, but a closer look reveals that we only need a finite set of information depending on the structures $\mathcal{A}$ and $\mathcal{A}^i$, and independent of Π. In step 2(b), for each constant of $\mathcal{A}^i$, we need to access its monadic memberships in $\mathcal{A}^c$. All we need in step 2(c), is for each interval and monadic relation the information whether this interval contains an element of this membership in $\mathcal{A}^c$. After the structure transformation, this finite set of information is known and can be hardwired into the algorithm. □

The program created by the algorithm in this lemma can be used to solve DATALOG-NONEMPTINESS, since it is an interval order type disjoint program over an interval sequence order with infinite intervals which are the prerequisites for Lemma 6.13. To solve DATALOG-NONEMPTINESS($\mathcal{A}$) we have to establish the connection between the nonemptiness of the IDBs of Π on $\mathcal{A}$ and the IDBs of its interval order type disjoint version Π' on $\mathcal{A}^i$.

Lemma 6.38 *Let $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ be a colored order, let $\varphi : A \to \mu^*$ be the mapping denoting the monadic memberships of the elements in A, where μ^* denotes all occurring monadic memberships in A and let $\mathcal{A}^i = (A_i, <, I_1, \ldots, I_s, c'_1, \ldots, c'_t)$ and $\mathcal{A}^c = (A_i, <, I_1, \ldots, I_s, M^c_1, \ldots, M^c_m, c'_1, \ldots, c'_t)$ be defined as in Definition 6.33.*

Let Π be a datalog program over $\mathcal{A}$ and Π' be the interval order type disjoint version of Π created by the algorithm in Definition 6.36.

Then for each IDB P of Π and the corresponding IDBs $\{P_1, \ldots, P_{n_P}\}$ of Π' it holds that:

$$P_\infty^{\Pi,\mathcal{A}^c} \neq \emptyset \iff \exists j \in \{1, \ldots, n_P\} \quad \text{with} \quad (P_j)_\infty^{\Pi',\mathcal{A}^i} \neq \emptyset \tag{6.6}$$

Proof: We will split the proof into two auxiliary lemmas, one for each implication of the claim and both working by induction on sequences of rule applications. The first lemma basically states that each instantiation of rules of Π is also an instantiation of some corresponding rules of Π'.

Lemma 6.39 *Let $(\rho_0, \ldots, \rho_i)$ be a sequence of rules of Π and for $j = 0, \ldots, i$ let $\bar{a}^{(j)}\bar{b}^{(j)}$ be an instantiation of ρ_j over $\mathcal{A}^c$ for which only the elements of earlier instantiations need to be present in the IDB relations occurring in the body of ρ_j.*

Then there is a sequence $(\rho'_0, \ldots, \rho'_i)$ of rules of Π', where for each j rule ρ'_j was created from ρ_j in Definition 6.36, and for each j, $\bar{a}^{(j)}\bar{b}^{(j)}$ is an instantiation of ρ'_j

over $\mathcal{A}^i$ for which only elements from earlier instantiations $\bar{a}^{(j')}\bar{b}^{(j')}$ with $j' < j$ are needed.

Proof: **Induction base:** In this case we have a rule ρ_0 whose body consists entirely of EDB atoms. This rule is in Definition 6.36 replaced by modified copies of this rule, each with different interval order type, for all interval order types that together with the order atoms and monadic atoms of ρ_0 are satisfiable. Since the instantiation $\bar{a}^{(0)}\bar{b}^{(0)}$ satisfies some interval order type τ, there is a rule ρ'_0 with this interval order type whose atoms are all satisfied by $\bar{a}^{(0)}\bar{b}^{(0)}$.

Induction step: In this case, the rule ρ_i contains IDB atoms in the body. By induction hypothesis, for each body IDB atom P there is an instantiation $\bar{a}^{(j')}\bar{b}^{(j')}$ with $j' < j$ adding some elements to the IDB P, which are then used in the instantiation $\bar{a}^{(i)}\bar{b}^{(i)}$ of ρ.

In Definition 6.36 the rule ρ_i is replaced by a modified copy of this rule, one for each satisfiable interval order type, there is a rule ρ'_i whose interval order type τ is satisfied by $\bar{a}^{(i)}\bar{b}^{(i)}$. Since the IDB atoms of this rule are satisfied by induction and the EDB atoms by τ, the tuple $\bar{a}^{(i)}\bar{b}^{(i)}$ is an instantiation of rule ρ'_i, as claimed. □

The second lemma has a similar approach as Lemma 6.38, creating for each sequence of instantiations a set of suitable replacement elements for elements from $\mathcal{A}^i$ which do not satisfy the monadic memberships required. In the following lemma we will use the following notation:

Definition 6.40 *Let $\bar{a} \in A^k$ and let $\mu \in (\mu^*)^k$. Let $\mu(a_i) := \mu_i$ for all i. For a tuple $\bar{b} \in A^\ell$ consisting of some or all entries of $\bar{a}$, we extend this definition to $\mu(\bar{b}) = (\mu(b_1), \ldots, \mu(b_\ell))$, where $\mu(b_j) = \mu(a_i)$, if b_j is the i-th entry of $\bar{a}$.*

Lemma 6.41 *Let $\xi \in \mathbb{N}$, let $\mu \in (\mu^*)^\xi$ be a vector of monadic memberships and let $\tau \in \Gamma_1(x_1, \ldots, x_\xi)$ be an order type. Let $\bar{c}$ denote the tuple consisting of the constants of $\mathcal{A}$ (which is equal to the tuple consisting of the corresponding constants of $\mathcal{A}^c$).*

Let $(\rho'_0, \ldots, \rho'_i)$ be a sequence of rules. For each $j \in \{0, \ldots, i\}$ let $\bar{a'}^{(j)}\bar{b'}^{(j)}$ be an instantiation of rule ρ'_j of Π' with elements of A^i such that

- *the instantiation $\bar{a'}^{(i)}\bar{b'}^{(i)}$ of rule ρ'_i only needs the elements from these preceeding instantiations in the IDB relations, and*
- *there is a tuple e' of elements from A^i such that the concatenation*

$$\bar{g'} := \bar{c}\bar{a'}^{(0)}\bar{b'}^{(0)} \cdots \bar{a'}^{(i)}\bar{b'}^{(i)}\bar{e'}$$

  of tuples satisfies τ.
- *For each entry h of $\bar{g}'$ equal to a constant of $\mathcal{A}^i$, $\mu(h) \preccurlyeq \varphi(h)$.*
- *For each entry h of $\bar{g}'$ from an interval I of $\mathcal{A}^i$, there exists an entry $h' \in I$ with $\mu(h) \preccurlyeq \varphi(h')$.*

Then it holds that:

There are rules $(\rho_0, \ldots, \rho_i)$ such that for all $j \in \{0, \ldots, i\}$, ρ'_j was created from ρ_j in Lemma 6.37. For each j with $0 \leq j \leq i$, there is an instantiation $\bar{a}^{(j)}\bar{b}^{(j)}$ of rule ρ_j of Π with elements of A such that

- *the instantiation $\bar{a}^{(i)}\bar{b}^{(i)}$ of rule ρ_i only needs the elements from these preceding instantiations in the IDB relations, and*
- *there is a tuple $\bar{e}$ of elements from A^c such that the concatenation*

$$\bar{g} := \bar{c}\bar{a}^{(0)}\bar{b}^{(0)} \cdots \bar{a}^{(i)}\bar{b}^{(i)}\bar{e}$$

of tuples satisfies τ and $\mu \preccurlyeq \varphi(\bar{g})$.

Proof: **Induction base:** The rule ρ'_0 does not contain IDB atoms in the body. The EDB atoms are order atoms and monadic atoms (with the interval EDBs).

We extend τ to τ' by adding all order atoms of ρ'_0. If in ρ'_0 there are multiple occurrences of a variable in the head IDB atom, then the corresponding tuple entries of the instantiation have to be equal and we add a corresponding equality atom to τ'; similar for constants in the head IDB atom of ρ'_0.

Let μ' be a vector of monadic memberships, which is created by from μ by component-wise union of the monadic memberships induced by the monadic atoms of rule ρ_i of program Π'.

Then $\bar{g}'$ satisfies τ', but the inequality $\mu' \preccurlyeq \varphi(\bar{g}')$ is not necessarily satisfied. We will now create a tuple $\bar{g}$, which satisfies both τ and the inequality $\mu \preccurlyeq \mu' \preccurlyeq \varphi(\bar{g})$.

For each entry f of $\bar{g}'$ equal to some constant of $\mathcal{A}^i$, by phase 2 of the algorithm in Definition 6.36, this element will satisfy the monadic membership induced by each rule of Π' and by the assumption of this lemma also $\mu(f) \preccurlyeq \varphi(f)$, hence $\mu'(f) \preccurlyeq \varphi(f)$.

The other entries must be from some intervals (which satisfy the properties of Lemma 6.34). Let I be an interval containing entries of $\bar{g}'$ and let $\bar{f}'$ be the tuple of these entries, ordered ascending. If $\bar{f}'$ satisfies $\mu'(\bar{f}') \preccurlyeq \varphi(\bar{f}')$, we set in $\bar{g}$ the corresponding entries to $\bar{f} := \bar{f}'$. Otherwise we have to create a set of replacement elements $\bar{f}$ from this interval with the same order type as $\bar{f}'$ and satisfying $\mu'(\bar{f}) \preccurlyeq \varphi(\bar{f})$.

By the assumption of this lemma, we have for each f'_j an element $h \in I$ with $\mu(f'_j) \preccurlyeq \varphi(h)$. Hence all elements of $\mu(\bar{f}')$ are contained in $\varphi(I)$. The monadic memberships in μ' induced by ρ_i are also present in I: During the creation of rule ρ'_i, in phase 2 this was checked. Since the rule was not discarded, for each entry f'_j of $\bar{f}'$ being part of the instantiation of ρ'_i there there is an element $h \in I$ with $\mu'(f'_j) \preccurlyeq \varphi(h)$. So for all monadic memberships from $\mu'(\bar{f})$, we have an element in I with this or greater membership.

Then there are also elements $\bar{f}$ in I ordered ascending and satisfying $\mu'(\bar{f}) \preccurlyeq \varphi(\bar{f})$ by Lemma 6.34. We set the corresponding entries of $\bar{g}$ to these entries $\bar{f}$.

For each interval of $\mathcal{A}^i$ we determine replacement elements this way and create $\bar{g}$. Since the intervals are disjoint and linearly ordered, $\bar{g}$ satisfies τ' and hence also τ. Furthermore $\mu \preccurlyeq \mu' \preccurlyeq \varphi(\bar{g})$.

It remains to verify that $\bar{a}^{(0)}\bar{b}^{(0)}$ is an instantiation of ρ_0. Since $\bar{g}$ satisfies τ', by construction of τ' all order atoms of ρ_0 are satisfied. By the construction of μ', all monadic EDB atoms of ρ_0 are satisfied by $\bar{a}^{(0)}\bar{b}^{(0)}$, hence $\bar{a}^{(0)}\bar{b}^{(0)}$ is indeed an instantiation of ρ_0.

Induction step:

The rule ρ_i contains IDB atoms in the body and may also contain EDB order atoms and monadic EDB atoms. As in the induction base, we extend τ to τ' and μ to μ' to contain all the monadic atoms of ρ_i and ensure that entries of $\bar{g}$ assigned to the same variable are equal. Additionally, we add equality atoms for all variables of ρ_i occurring in IDB atoms of the body of ρ: If a variable x is assigned an entry a of $\bar{a}^{(i)}\bar{b}^{(i)}$ in the instantiation of ρ_i and x also occurs in an IDB atom Q, the tuple used from the relation Q was added in some previous instantiation $\bar{a}^{(j)}\bar{b}^{(j)}$ with $j < i$ and this instantiation contains an entry b corresponding to x in ρ_i. We add an atom $a = b$ to τ' (where a and b denote the entries of $\bar{g}$). These atoms of τ' ensure that each satisfying tuple $\bar{g}$ contains a consistent variable assignment of ρ_i. Then $\bar{g}$ satisfies τ'.

During the creation of ρ_i' in phase 2 of the algorithm it is assured that μ' and $\bar{g}'$ satisfy the last two properties of the assumption of this lemma.

We now apply the induction hypothesis for $i-1$ on the tuple $\bar{g}'$. Here the instantiation $\bar{a'}^{(i)}\bar{b'}^{(i)}$ of ρ_i' in the i-th stage is considered as part of the tuple $\bar{e}'$: The induction hypothesis is applied to:

$$\bar{g}' := \bar{c}\bar{a'}^{(0)}\bar{b'}^{(0)} \cdots \bar{a'}^{(i-1)}\bar{b'}^{(i-1)}(\bar{e^*})'$$

with $(\bar{e^*})' = \bar{a'}^{(i)}\bar{b'}^{(i)}\bar{e}'$.

By the induction hypothesis we have a tuple

$$\bar{g} := \bar{c}\bar{a}^{(0)}\bar{b}^{(0)} \cdots \bar{a}^{(i-1)}\bar{b}^{(i-1)}\bar{e^*}$$

with the properties from the claim of this lemma. Using the arity of $\bar{a'}^{(i)}\bar{b'}^{(i)}$ we can split $\bar{e^*}$ into:

$$\bar{e^*} = \bar{a}^{(i)}\bar{b}^{(i)}\bar{e}$$

Then the tuple $\bar{g}$ satisfies τ' and hence also $\tau \subseteq \tau'$, and satisfies $\mu \leq \mu' \preccurlyeq \varphi(\bar{g})$. We only have to show that $\bar{a}^{(i)}\bar{b}^{(i)}$ is an instantiation of ρ_i.

By the construction of τ', $\bar{a}^{(i)}\bar{b}^{(i)}$ is a consistent variable assignment of ρ and it satisfies all order EDBs of ρ_i. By the choice of μ', all monadic atoms of ρ_i are satisfied as well.

Now consider an IDB atom of ρ_i. By the construction of τ', the entries of $\bar{a}^{(i)}\bar{b}^{(i)}$ assigned to variables in this are equal to some $\bar{a}^{(j)}$ with $j < i$. Hence this IDB atom is also satisfied.

Since all atoms of ρ_i are satisfied, $\bar{a}^{(i)}\bar{b}^{(i)}$ is indeed an instantiation of ρ_i as claimed.

□

With these two lemmas, the two implications of Lemma 6.38 are shown. □

Now we have all tools for an implementation of the scheme from Remark 6.14 and derive our main result of this chapter:

Theorem 6.42
On linear orders $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_r)$ with finitely many constants (which may be used by the datalog programs) and finitely many monadic relations DATALOG-NONEMPTINESS($\mathcal{A}$) can be decided in exponential time.

Proof: The problem instance consists of a program Π on $\mathcal{A}$ and an IDB P of Π.

Lemma 6.35 shows that there is a colored interval sequence order $\mathcal{A}^c$ such that the nonemptiness of the IDB relations of each program Π on $\mathcal{A}$ and $\mathcal{A}^c$ is equivalent. In Lemma 6.37 a transformation algorithm is given which transforms any program Π on $\mathcal{A}$ (and hence $\mathcal{A}^c$) to an equivalent interval order disjoint program Π' on an interval sequence order $\mathcal{A}^i$ in time at most exponential in $|\Pi|$ and the length of Π' is at most exponential in $|\Pi|$. Lemma 6.38 shows this equivalence by showing that an IDB P of Π on $\mathcal{A}^c$ is nonempty if and only if one of the corresponding IDBs $\{P_1, \ldots, P_{n_P}\}$ created in Lemma 6.37 is nonempty, where n_P is at most exponential in $|\Pi|$.

Since Π' is an interval order disjoint program on an interval sequence order $\mathcal{A}^i$ and by Lemma 6.34 the intervals of $\mathcal{A}^i$ are all infinite, we can compute an initialization sequence for Π' on $\mathcal{A}^i$ by Lemma 6.13. This computation run is time polynomial in $|\Pi'|$ (hence exponential in $|\Pi|$) and after this computation the nonemptiness of each IDB of Π' is known. With this information we can answer the question whether the IDB P of Π is nonempty. □

For solving DATALOG-NONEMPTINESS($\mathcal{A}$), we do not actually need to change the representation of the structure $\mathcal{A}$ when creating the intervals, but we rather derive a modification scheme for datalog programs, the algorithm in Lemma 6.37. So no new representation issues for infinite structures arise.

6.4 Negation in Datalog Rules

Again, we consider the changes introduced by datalog$^\neg$, where negation may occur in front of EDB atoms.

While the effects of negating order atoms are the same as in Chapter 5 (only some program transformation), leading to an exponential increase of the upper bounds, negation of monadic relations does not change the complexity at all. For each monadic relation M_1, we simply add its complement $M_2 := A \setminus M_1$ to our structure and may replace in the datalog program any occurrence $\neg M_1(x)$ by $M_2(x)$. A close look at the methods for solving DATALOG-NONEMPTINESS($\mathcal{A}$) shows that the emptiness of the IDB relations of the modified program can be determined in exponential time.

So the negations important for the running time of solving DATALOG-NONEMPTINESS are the negations in front of order atoms. Programs containing such negations can be translated to (exponentially larger) programs without negations as shown in Chapter 5, leading to:

Lemma 6.43 *DATALOG-NONEMPTINESS($\mathcal{A}$) for datalog$^\neg$ programs (with EDB negations) on colored orders $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_k)$ with constants is solvable in deterministic doubly exponential time, hence in* 2EXP.

6.5 Colored Orders and DATALOG-TUPLE

So far, we have excluded DATALOG-TUPLE from our discussion in this chapter. In Chapter 5, we simply added DATALOG-TUPLE as an extra step by first solving DATALOG-NONEMPTINESS($\mathcal{A}$) and then calculating the full fixed point. In the types describing the fixed point stage of all IDB relations, we could check then whether a tuple satisfied one of the types and hence was part of the relation. The naive idea of applying this approach to the scenario presented in the current chapter does not lead to a solution of DATALOG-TUPLE.

In the construction of the disjoint intervals and hence in the construction of the disjoint IDB sets, we removed some elements which could be replaced by elements satisfying more monadic membership constraints. But the tuple $\bar{a}$ from our DATALOG-TUPLE instance may just contain some of the removed elements in which case entries of the tuple and also the tuple itself are not in the structure $\mathcal{A}'$. Simply adding these elements in later steps may lead to a wrong solution, since adding these elements destroys the connection between intervals and monadic membership constraints, which is used in Theorem 6.42 to solve the DATALOG-NONEMPTINESS.

A workaround for this problem is to keep the elements of the tuple $\bar{a}$ in the structure by the use of additional constants. Adding one constant for each value that occurs as an entry of the tuple $\bar{a}$, we ensure that these values will survive all changes from $\mathcal{A}$ to $\mathcal{A}'$. If our DATALOG-TUPLE($\mathcal{A}$) instance is of form $(\Pi, P, \bar{a})$, we transfer it to a DATALOG-NONEMPTINESS($\mathcal{A}'$) instance (Π', P') by adding the following rule

$$P' \leftarrow P(\bar{a}).$$

representing $\bar{a}$ by the constants introduced. Then solving the nonemptiness of (Π', P') solves DATALOG-TUPLE($\mathcal{A}$) instance $(\Pi, P, \bar{a})$, since

$$(P')_\infty^{\Pi',\mathcal{A}} \neq \emptyset \quad \Longleftrightarrow \quad \bar{a} \in P_\infty^{\Pi,\mathcal{A}}$$

So why not use this approach to solve DATALOG-TUPLE($\mathcal{A}$) with a low uniform time bound? A serious problem is that we cannot guarantee any running time bounds in this case. Depending on the representation of $\mathcal{A}$ and the tuple entries of $\bar{a}$, the running time may increase a lot, much more than, e.g., the singly exponential time bound.

Example 6.44 *Consider the following substructure $\mathcal{A} = (A, <)$ of the rational numbers $(\mathbb{Q}, <)$ with the following properties:*

- *A contains all integers, and each integer i is represented by a bit string of length $\sim \log_2 i$.*
- *Between each two integers i and $i+1$, A contains all rational numbers representable with 2^i digits.*

Although we omit monadic relations or constants here, we will consider what happens if we use the above approach to solve the DATALOG-TUPLE($\mathcal{A}$).

As tuple we consider $\bar{a} = (a_1, a_2) := (i, i+1)$, i.e. two consecutive integers, part of our DATALOG-TUPLE($\mathcal{A}$) instance $(\Pi, P, \bar{a})$. As assumed, we can encode this tuple using $O(\log_2 i)$ bits.

If we convert a_1 and a_2 into constants, we have an interval $(a_1, a_2) = (i, i+1)$ containing 10^{2^i} elements, which is a finite quantity and will be converted to the same number of constants by our solving method. When transforming the program Π to a program with pairwise distinct interval and color types, we will have to add a number of rules, which is exponential in 10^{2^i}, in other words triply exponential in i. Ignoring the other parts of the input, even i itself is exponential in the length of $\bar{a} = (a_1, a_2) = (i, i+1)$, which is part of the input.

This leaves no chance for a program which is only exponential bigger, destroying the singly exponential upper bound of our algorithm in this case.

This example shows that even without any monadic relation which could possibly increase the complexity of the problem, the approach presented in this chapter is unsuitable for solving DATALOG-TUPLE.

If we plan to use this approach aiming at the decidability of DATALOG-TUPLE($\mathcal{A}$) for a colored order, we have the problem that the newly added constants (of the input tuple $\bar{a}$) are part of the input and we do not know the whole structure in advance to create the algorithm building an interval order type disjoint version program (see Lemma 6.37) using the structure transformation of Definition 6.33 implicitly. Since a part of the structure is contained in the input, the solving algorithm would have to carry out the transformation in Definition 6.33 as a first phase and then create the algorithm to modify the datalog program. For that, each property of the enriched structure needed in this Definition 6.33 would have to be decidable, including for example for all given elements $a, b \in A$ the question if there is an infinite sequence of cycling monadic memberships (condition *P1* of Lemma 6.30) in the interval (a, b) and if not how the finite partition of the interval (a, b) with condition *P2* of Lemma 6.30 looks like.

If the representation of a colored order $\mathcal{A}$ would allow to decide all these questions in finite time then DATALOG-TUPLE($\mathcal{A}$) would be decidable.

Unlike the case of linear orders (without colors), we can show that for arbitrary colored orders $\mathcal{A}$ DATALOG-TUPLE($\mathcal{A}$) cannot be in EXPTIME (neglecting representation issues). In the following example we show a stronger result: For an arbitrary

structure $\mathcal{A} = (A, M)$ having a monadic relation DATALOG-TUPLE($\mathcal{A}$) is not in EXPTIME.

Let $\mathcal{C}$ be a time-complexity class, defined as $\mathcal{C} = \bigcup_{f \in F} \mathrm{DTIME}(f)$ for some set F of complexity functions. For example, for $\mathcal{C}$ =PTIME, F is the set of polynomials, and for $\mathcal{C}$ =3EXP, $F = \left\{ 2^{2^{2^{n^k}}} \;\middle|\; k \in \mathbb{N} \right\}$.

Example 6.45 *Let $\mathcal{A} = (\mathbb{N}, M)$ the set of natural numbers together with a monadic relation M containing the encoding of all starting configurations x of runs of a universal Turing machine, which accept after at most $f(|x|)$ steps for some $f \in F$.*

Then DATALOG-TUPLE($\mathcal{A}$) is hard for $\mathcal{C}$, as we see using the following simple program:

$$P(x) \leftarrow M(x).$$

Any problem instance y in $\mathcal{C}$ can be solved by a Turing machine in time $f(|y|)$ for some $f \in F$, consequently also by the universal Turing machine. This leads to a starting configuration x of a universal Turing machine, of which we want to know if it leads to an accepting configuration (in time $f(|x|)$ for some $f \in F$). This can be checked by a test if it is included in M and or equivalently in the relation $P_\infty^{\Pi,\mathcal{A}}$.

To see that DATALOG-TUPLE($\mathcal{A}$) for this structure is not in EXPTIME, we may choose $\mathcal{C}$ =3EXP with F as above. Moreover, we can incorporate any complexity function in M, leading to arbitrarily high complexity, even undecidability, if we drop the constraint that our structures have to be decidable. Note that we did not even employ the usage of an order, but adding it could clearly only increase the complexity, not decrease it.

6.6 Summary of this Chapter

We first transferred the concept of distance types to colored orders, leading to interval order and distance types, and showed that there is an initialization sequence for deciding DATALOG-NONEMPTINESS, similar to non-colored orders.

For the solution of DATALOG-NONEMPTINESS, we always derive a nice form of the structure (without actually creating it) to create a solving algorithm that transforms the datalog program to some interval-type-disjoint form corresponding to this structure and then solves the nonemptiness using an initialization sequence. The modified version of the structure allows us to encode monadic memberships within intervals, leading to the main result of this chapter:

Theorem 6.42

> On linear orders $\mathcal{A} = (A, <, M_1, \ldots, M_m, c_1, \ldots, c_r)$ with finitely many constants (which may be used by the datalog programs) and finitely many monadic relations DATALOG-NONEMPTINESS($\mathcal{A}$) can be decided in exponential time.

We also used the approach presented in this chapter to derive the EXPTIME upper bound for DATALOG-NONEMPTINESS on linear orders with constants mentioned in Chapter 5.

Datalog on Orders and Relation Arities

The classification of orders with additional EDB relations in this work is exhaustive: In Chapter 5 no additional relations are allowed, leading to EXPTIME complexity. In this Chapter, additional monadic EDBs are allowed, leading to EXPTIME complexity. Chapter 4 shows that if arbitrary additional relations of arity two are allowed, the DATALOG-NONEMPTINESS is undecidable by using a successor structure as EDB. Of course, the latter case includes the complexity of DATALOG-NONEMPTINESS for orders with additional EDBs of higher arity.

The method introduced in Chapter 5 shows how to solve DATALOG-NONEMPTINESS($\mathcal{A}$) for datalog$^{\neg}$, which we have used for colored orders as well, noting that negations of monadic EDB relations do not raise upper bounds further.

Open problems

While we will see an application of the methods of this chapter in Chapter 7 (Tree Orders), the method seems promising for various structures, leading to the obvious question:

Question 6.1 *What other structures allow similar type concepts and type disjoint programs?*

Although the type concept introduced in this chapter and the preceeding chapter seem suitable to show upper bounds, there may be other approaches leading to upper bounds for datalog, maybe even given a structural answer to the following question:

Question 6.2 *On which structures does DATALOG-NONEMPTINESS have EXPTIME complexity, or is at least decidable?*

Chapter 7

Datalog on Tree Orders

The orders considered in this work so far have all been linear orders and we used the property that these orders are total. Not all orders of interest are linear orders. Considering for example vertices in graphs or sets, partial orders are of interest and datalog on partial orders can be used for the description and computation of some properties of partial orders.

In this chapter we will not give an answer to the question what computational power and expressibility datalog possesses on arbitrary partial orders, but we rather apply the methods introduced in the previous chapters to some partial orders. Our methods rely on the comparison of elements by order atoms. While arbitrary partial orders have the disadvantage that there may be incomparable elements a and b, for which neither a common greater element d (i.e. $a < d$ and $b < d$) nor a common smaller element c (i.e. $c < a$ and $c < b$) exists, we focus on tree orders. In tree orders, for incomparable elements a and b there always exists a common smaller element and there never exists a common greater element. This enables us to group these elements by their common smaller element and yet handle all elements greater than a and all elements greater than b separately, since they are incomparable for incomparable a and b.

This chapter shows an application of methods introduced earlier: From Chapters 5 and 6, we borrow the concept of type disjoint programs which we mainly use here in the version of Chapter 6 for colored orders. Although we do not have colors here, the transformation of structures into disjoint intervals — deriving a transformation algorithm for datalog programs into type disjoint versions — is extended to work for trees as well.

This chapter is meant to be a brief application and transfer of the methods introduced earlier which is why we only sketch some proofs that are similar to the proofs in earlier chapters.

Outline of this Chapter

After defining tree orders and notations we derive some upper bounds for tree orders without constants, followed by the corresponding results about tree orders with constants. As a variant, we leave the discreteness of infinite trees behind and allow dense parts in some restricted manner of appearance. We finish the chapter by a look at other tree order variants and a discussion about DATALOG-TUPLE in this context.

7.1 Preliminaries

Infinite trees are a natural extension of finite trees and have been introduced in a graph theoretic context (see, e.g., [Die00]). The usual notion of levels in a tree, sometimes called height or depth of a vertex in a tree, leads to a concept of partial ordering on trees, namely the ordering of the vertices on paths in the tree by their level. But infinite trees can also be defined the other way round, starting with a partial order, and restricting it to tree shape (see, e.g. [Hod97]).

7.1.1 Tree Orders

Definition 7.1 ***(Partial Order)***
A partial order $\mathcal{A} = (A, <)$ *consists of a set* A *and a relation* $<^{\mathcal{A}}$ *which is* transitive *and* antireflexive[1].

Let $a \in A$*, then we define:*

$$a^{>} = \{b \in A \mid a < b\}$$
$$a^{<} = \{b \in A \mid b < a\}$$

A chain *is a linearly ordered subset of* A.

We call two elements $a, b \in A$ comparable*, if* $a < b$*,* $a = b$ *or* $a > b$ *holds, otherwise* incomparable.

Definition 7.2 ***(Tree-Like Order)***
A tree-like order $\mathcal{A} = (A, <)$ *is a partial order satisfying:*

- *For all* $a \in A$ *the set* $a^{<}$ *is a finite chain.*
- *There is an* $r \in A$ *such that for all* $a \in A$*,* $r < a$ *or* $a = r$*.*

Lemma 7.3 *Let* $\mathcal{A} = (A, <)$ *be a tree-like order and let* $a, b \in A$ *be incomparable. Then* $a^{>}$ *and* $b^{>}$ *are disjoint.*

Proof: Assume, that there exists a $c \in a^{>} \cap b^{>}$. Then $c^{<}$ is a chain including both a and b. Hence a and b are comparable. □

[1] We keep orders strict in this chapter as well.

Definition 7.4 ***(Tree Order)***
Let $\mathcal{A} = (A, <)$ be a tree-like order and let $a \in A$. The successors of a are defined as:

$$S(a) = \{b \in a^{>} \mid \text{there is no } c \in A \text{ with } a < c < b\}$$

A tree-like order $\mathcal{A} = (A, <)$ is a tree order*, if for all $a \in A$ the set $S(a)$ is finite.*

For a better description of tree orders and for using them within the creation of the program transformation algorithms, we need some additional notation.

Definition 7.5 ***(Distance in a Tree Order)***
Let $\mathcal{A} = (A, <)$ be a tree order, $a, b \in A$ with $a < b$. $C := \{c \in A \mid a < c < b\}$ is a finite chain. We define the distance of a and b as $d(a, b) := |C| + 1$, where $|C|$ is the cardinality of C. For all $a \in A$, we let $d(a, a) = 0$.

Definition 7.6 ***(Branching Point)***
Let $\mathcal{A} = (A, <)$ be a tree order and $a, b \in A$ incomparable. Both $a^{<}$ and $b^{<}$ are finite chains and $r \in a^{<} \cap b^{<}$. Then the branching point of a and b is the following well defined element:

$$\mathrm{bp}(a, b) := \max(a^{<} \cap b^{<})$$

An element $c \in A$ is a branching point if there are incomparable $a, b \in A$ with $c = \mathrm{bp}(a, b)$. The set of all branching points of a set $S \subseteq A$ is denoted by $\mathrm{bp}(S)$.

The branching points of $\mathcal{A} = (A, <)$ can alternatively be characterized as the elements with more than one successor.

The reason for calling this kind of partial orders "tree orders" can be motivated by the following connection:

Remark 7.7 *Let $\mathcal{A} = (A, <)$ be a tree order. Then $T = (A, E)$ with relation E defined as*

$$(u, v) \in E \quad \Longleftrightarrow \quad \text{there is a } u \in A \text{ and a } v \in S(u)$$

is a (finite or infinite) directed tree with root vertex r and with finite branching, i.e. each vertex has only finitely many children.

7.1.2 DATLOG-NONEMPTINESS and Homomorphisms

In this chapter we will transform tree orders into a shape more suitable for solving DATALOG-NONEMPTINESS by removing some parts of the structure. To show that DATALOG-NONEMPTINESS is equivalent, we will define a homomorphism from the original structure to the modified substructure and use the following fact (see Remark 2.3 in Chapter 2):

Let $\mathcal{A}$ and $\mathcal{B}$ be two structures over the same vocabulary τ. Let Π be a datalog program over $\mathcal{A}$ and let $h : \mathcal{A} \to \mathcal{B}$ be a homomorphism. Then for each IDB P of Π and each tuple $\bar{a}$ over $\mathcal{A}$ with the same arity as P it holds that:

$$\bar{a} \in P_\infty^{\Pi,\mathcal{A}} \qquad \Longrightarrow \qquad h(\bar{a}) \in P_\infty^{\Pi,\mathcal{B}}$$

7.2 Upper Bound for Tree Orders

By König's tree lemma (see Lemma 5.22), each tree order $\mathcal{A} = (A, <)$ falls into exactly one of the following two categories, which will determine our choice for the algorithm to solve DATALOG-NONEMPTINESS($\mathcal{A}$):

1. $\mathcal{A}$ contains an infinite chain C.
2. $\mathcal{A}$ is finite.

The EXPTIME-completeness of DATALOG-NONEMPTINESS($\mathcal{A}$) is well known for finite structures $\mathcal{A}$ (see, e.g., [DEGV97]), and so we only have a look at the first case. The main observation here is that a datalog program as a positive existential formalism is not able to force two elements $a, a' \in A$ used in the program to be incomparable.

Lemma 7.8 *Let $\mathcal{A} = (A, <)$ be a tree order containing an infinite chain C. Then DATALOG-NONEMPTINESS($\mathcal{A}$) and DATALOG-NONEMPTINESS($\mathcal{C}$) are equivalent where $\mathcal{C} = (C, <)$ with the order $<$ restricted from A to C.*

Proof: One direction of the proof is trivial using the embedding homomorphism of $\mathcal{C}$ into $\mathcal{A}$.

For the other direction we define a homomorphism from $\mathcal{A}$ to $\mathcal{C}$ using for each element the distance from the root element r, which is well defined.

Since $\mathcal{C}$ is a chain in a tree order, for each $i \in \mathbb{N}$ there is a unique element $c \in C$ with $d(r, c) = i$. We define a mapping $h : A \to C$ as follows:

Let $a \in A$. Then we let $h(a) = c$, where $c \in C$ is the unique element with $d(r, a) = d(r, c)$.

This mapping h is a homomorphism from $\mathcal{A}$ to $\mathcal{C}$: Let $a, b \in A$ with $a <^{\mathcal{A}} b$. Then $d(r, a) < d(r, b)$ and by the definition of h: $d(r, h(a)) < d(r, h(b))$. Since $\mathcal{C}$ is a chain, $h(a) <^{\mathcal{C}} h(b)$ follows. □

Since this is the only infinite case to consider here, we may solve DATALOG-NONEMPTINESS on tree orders $\mathcal{A} = (A, <)$:

Theorem 7.9 *On tree orders $\mathcal{A} = (A, <)$, DATALOG-NONEMPTINESS($\mathcal{A}$) is EXPTIME-complete.*

Proof: The lower bound for strict orders follows from Corollary 3.3.

For the upper bound, we consider two cases: Either $\mathcal{A}$ is finite, in which case the EXPTIME-upper is a known result (see [DEGV97]). Or $\mathcal{A}$ is infinite and König's tree lemma (Lemma 5.22) guarantees the existence of an infinite chain C in $\mathcal{A}$. Using Lemma 7.8, we transfer the problem solution of DATALOG-NONEMPTINESS to the infinite chain C which is a linear order. Results from Chapter 5 show how to solve DATALOG-NONEMPTINESS(C) and hence also DATALOG-NONEMPTINESS($\mathcal{A}$) in EXPTIME. □

7.3 Upper Bounds for Tree Orders with Constants

We will describe a procedure for infinite tree orders with constants. This procedure also works for finite tree orders, and moreover, applied to finite tree orders, it is similar to the solution method for finite datalog programs which inserts all possible combinations of elements into the variables, generating a ground version of the program (see, e.g., [DEGV97]).

As in Chapter 6, we will employ the scheme in Remark 6.14 to solve DATALOG-NONEMPTINESS($\mathcal{A}$) on tree orders. In this scheme we first analyze the structure $\mathcal{A}$ deriving a transformation to create an algorithm solving the problem. This algorithm transforms the program of the problem instance according to the structure transformation and then solves the nonemptiness using an initialization sequence. As in Chapter 6, no changes to the representation of the structure $\mathcal{A}$ are actually carried out, but we collect all these changes as information about the structure in the solving algorithm.

As the proofs of these concepts are similar to the proofs used in Chapter 6, we will only give sketches of proofs showing the principles and tools needed.

We first describe the special shape of tree orders we create to solve DATALOG-NONEMPTINESS (an adaptation of the interval sequence orders introduced in Section 6.1.2).

Definition 7.10 *Let $\mathcal{A} = (A, <, c_1, \ldots, c_k)$ be a tree order with constants. Two constants c_i and c_j are called* adjacent *if*

- *c_i and c_j are comparable* and
- *there is no constant between them (w.r.t. $<$).*

For two adjacent constants c_i and c_j with $c_i < c_j$ the bounded constant interval *(c_i, c_j) is the set*

$$(c_i, c_j) := (c_j^< \setminus c_i^>) \setminus \{c_i\} \;=\; \{x \in A \;\mid\; c_i < x < c_j\} \quad .$$

Let c_i be a constant for which no bigger constant c_j exists, i.e. no constant c_j with $c_i < c_j$, and for which the set $c_i^> = \{c \in A \;\mid\; c_i < x\}$ is linearly ordered. Then $c_i^>$ is an unbounded constant interval.

A constant interval *is a* bounded interval *or an* unbounded interval.

Definition 7.11 $\mathcal{A} = (A, <, c_1, \ldots, c_k)$ *is a* tree order with empty or infinite constant intervals *if*

1. $\mathcal{A}$ *is a tree order and*
2. A *is partitioned into the constants and constant intervals, i.e. each* $a \in A$ *is equal to a constant or contained in a constant interval, and*
3. *all bounded constant intervals are empty.*

This is the shape of tree order we will try to achieve to solve DATALOG-NON-EMPTINESS($\mathcal{A}$). In the following lemma we sketch a method to derive a tree order with empty or infinite constant intervals on which the nonemptiness of datalog IDB relations stays unchanged.

Lemma 7.12 *Let* $\mathcal{A} = (A, <, c_1, \ldots, c_k)$ *be a tree order. Then there exists a tree order* $\mathcal{B}$ *with empty or infinite constant intervals which can be created from* $\mathcal{A}$ *with finitely many changes and such that for each datalog program* Π *on* $\mathcal{A}$ *and each IDB* P *of* Π *the following holds:*

$$P_\infty^{\Pi,\mathcal{A}} \neq \emptyset \quad \Longleftrightarrow \quad P_\infty^{\Pi,\mathcal{B}} \neq \emptyset$$

Proof: (Sketch)

We give the transformations leading from $\mathcal{A}$ to $\mathcal{B}$ and for each transformation we sketch how it can be shown that the nonemptiness of IDB relations does not change.

Step 1: Finite Part of $\mathcal{A}$ Containing the Constants

One main observation making things easier is that on tree orders, as we have defined them, constants may only appear on elements having a finite distance to the root element, which limits constants to a finite part of the structure.

For each constant c_j, the set $c_j^< = \{x \in A \mid x < c_j\}$ is finite, hence also the following set F is finite:

$$F := \bigcup_{j=1}^{k} c_j^<$$

We now enhance the structure by introducing a new constant for each element $f \in F$ which is not equal to a constant.

Adding constants which are not used in Π does not change the IDB relations.

Step 2: Transforming $c_i^>$ into Infinite Intervals

Consider each constant c_i which is a branching point and which has a successor $s \in S(c_i)$ not equal to a constant. We collect all elements greater than such a successor s in a set S_n:

$$S_n := \{b \in a^> \mid a \in S(c_i) \text{ and } a \text{ is } \textbf{not} \text{ equal to some constant } c_j\}$$

Let C be a longest chain in S_n. We replace S_n by C and add a new constant c_j pointing to the one element in $C \cap S(c_i)$. Note that this new constant is *not* a branching point.

The set $c_j^>$ is clearly an unbounded interval. All successors of c_i are now equal to constants, hence the intervals bounded from below by c_i are bounded empty intervals.

The nonemptiness of IDB relations is not affected by this change: Let $\mathcal{A}'$ be the structure before this change and $\mathcal{A}''$ the changed structure. We define a mapping $h : \mathcal{A}' \to \mathcal{A}''$ as follows:

Outside the set S_n, h is the identity mapping. On S_n, h copies the distance from c_i: Let $a \in S_n$. Since C is a chain of sufficient length, there is a unique element $c \in C$ with $d(c_i, a) = d(c_i, c)$ and we let $h(a) = c$. Then h is a homomorphism on S_n with a distance argument as in the proof of Lemma 7.8, and also on the rest of the structure by definition.

Only the finitely many constants present after Step 1 are branching points, since newly introduced constants are always lower bounds of an interval, leading to a finite total number of replacements.

Step 3: Adding Constants to Remove Finite Intervals

Similar to the methods of Chapter 6, the last step consists of placing constants on any elements in finite chains which are not yet covered by constants, eliminating finite substructures without constants. No constants introduced in this step are branching points, since only finite linearly ordered sets are filled with constants in this step.

□

In this section we use a replacement for the distance type concept introduced in Chapters 5 and 6. To be able to create an initialization sequence, we use the following tool which gives us the necessary information about tuple entries in unbounded constant intervals to solve DATALOG-NONEMPTINESS using an initialization sequence.

Lemma 7.13 *Let $\mathcal{A} = (A, <, c_1, \ldots, c_k)$ be a tree order with empty or infinite constant intervals and let Π be a program over $\mathcal{A}$. Let P be an IDB of Π, $i \in \mathbb{N}$ and $\bar{a}$ be a tuple with $\bar{a} \in P_i^{\mathcal{A},\Pi} \setminus P_{i-1}^{\mathcal{A},\Pi}$. Let $c_j^>$ be an unbounded constant interval and let $\bar{b}$ the tuple of entries of $\bar{a}$. Let τ be the complete distance type satisfied by $(c_j\bar{b})$ in which each distance atom has the maximal rank satisfied by $(c_j\bar{b})$.*

Let $\bar{e}$ be a tuple with the same arity as $\bar{b}$ and entries in $c_j^>$ such that $(c_j\bar{e})$ satisfies τ. Let $\bar{d}$ be the tuple created from $\bar{a}$ by replacing the entries $\bar{b}$ by $\bar{e}$. Then it holds that: $\bar{d} \in P_i^{\mathcal{A},\Pi}$.

Proof: (Sketch)

The claim can be shown by induction over the rule applications:

In the induction base, the tuple $\bar{a}$ is added using a rule ρ containing only order atoms and the entries $\bar{b}$ are satisfying some of these atoms. If we replace $\bar{b}$ by a tuple $\bar{c}$ satisfying τ, all these atoms are also satisfied by these new entries. Since the other entries of $\bar{d}$ are the same as in $\bar{a}$ and there are no order atoms between elements of different unbounded constant intervals, all atoms of ρ are satisfied by $\bar{d}$ and the claim follows.

In the induction step, the tuple $\bar{a}$ is added using a rule ρ with IDB and EDB atoms. There are two cases to consider: If the entries $\bar{b}$ of $\bar{a}$ in $c_j^>$ are not assigned to variables occurring in IDB atoms, the claim follows exactly as in the induction base. If on the other hand entries of $\bar{b}$ correspond to variables in IDB atoms, we apply the induction hypothesis showing that each tuple containing elements of $c_j^>$ is also contained in the corresponding IDB relation. Since this holds for all IDB atoms, the claim depends only on the EDB atoms of the rule and for them the claim follows as in the induction base. □

For an initialization sequence we need a type disjoint version of the program and for this we need a type concept which is similar to the interval types introduced in Chapter 6, but simpler.

Definition 7.14 *Let $\mathcal{A} = (A, <, c_1, \ldots, c_k)$ be a tree order with empty or infinite constant intervals. Let $\bar{x} = (x_1, \ldots, x_m)$ be a variable tuple. A* complete tree interval type *$\tau = (\gamma, \delta)$ is a order type γ and a set δ of atomic formulae of the following kind:*

- *For each variable x_i there is either a formula $x_i = c_j$ for some j in δ or a formula $x_i > c_j$ in δ where $c_j^>$ is an unbounded interval.*
- *For each pair of variables $(x_i, x_{i'})$ for which a j exists with $(x_i > c_j) \in \delta$ and $(x_{i'} > c_j) \in \delta$ there is an order atom containing both x_i and $x_{i'}$ in γ.*

A tree interval type is incomplete, if for some variables some of these formulae are missing.

A type disjoint program is then defined as:

Definition 7.15 *A datalog program over a tree order $\mathcal{A}$ with empty or infinite constant intervals is* tree interval type disjoint *if for every IDB P there is a complete tree interval type which all tuples in P_∞^Π have to satisfy.*

Lemma 7.16 *Let $\mathcal{A} = (A, <, c_1, \ldots, c_k)$ be a tree order with empty or infinite intervals. Then there is an algorithm which creates from each datalog program Π over $\mathcal{A}$ an equivalent tree interval type disjoint datalog program Π' with the following property:*

For every IDB P of Π there are a number n_P and IDBs $P_1, \ldots, P_{n_P}$ of Π' of pairwise distinct tree interval types such that

$$P_\infty^{\Pi,\mathcal{A}} = \bigcup_{j=1}^{n_P} (P_j)_\infty^{\Pi',\mathcal{A}}.$$

The running time of this algorithm is in $O(d^{|\Pi|})$ for some suitable constant d depending on $\mathcal{A}$ and also $|\Pi'| \in O(d^{|\Pi|})$. We call the program Π' created by this algorithm the tree interval type disjoint version of Π.

Proof: (Sketch)

The proof of this lemma can be carried out similarly to the proofs of Lemmas 5.12 and 6.37. The only difference here is that the partial nature of the order $\mathcal{A}$ limits the number of possibilities: If a variable x is already constrained to be $c < x$, then constants and hence intervals incomparable with c do not have to be considered here.

The algorithm runs mainly in two phases: The first creates copies of the rules to equip them with complete tree interval types and the second renames the IDBs accordingly.

1. Let $\{I_1, \ldots, I_m\}$ be the unbounded constant intervals of $\mathcal{A}$.

2. For each rule ρ of Π:

   Let $x_1, \ldots, x_\ell$ be the variables of ρ. Let $C = \{I_1, \ldots, I_m, c_1, \ldots, c_k\}$. For each $\bar{\xi} \in C^\ell$ and each complete order type $\gamma \in \Gamma_1(x_1, \ldots, x_\ell)$:

   If for some i, j, $\{(x_i < x_j), (x_i = x_j)\} \cap \gamma \neq \emptyset$ and $\xi_j < \xi_i$, drop this combination $\gamma, \bar{\xi}$ and continue with the next. If for some i, j, ξ_i and ξ_j are incomparable, delete the atoms with x_i and x_j from γ.

   We add a rule $\rho_{\bar{\xi},\gamma}$ to Π' created from ρ with the following modifications:

   (a) If $\xi_i = I_j$ for some i, j, then there is a j' such that $I_j = c_{j'}$. Add $c_{j'} < x_i$ to $\rho_{\bar{\xi},\gamma}$.
   (b) If $(x_i < x_j) \in \gamma$ for some i, j and $\xi_i = \xi_j$, add $x_i < x_j$ to $\rho_{\bar{\xi},\gamma}$.
   (c) If $\xi_i = c_j$ for some i, j, replace all occurrences of x_i by c_j in $\rho_{\bar{\xi},\gamma}$.
   (d) If $(x_i = x_j) \in \gamma$ for some i, j, replace all occurrences of x_i by x_j in $\rho_{\bar{\xi},\gamma}$ (or by the corresponding variable or constant, if x_j has been replaced earlier).

3. The complete interval order type of the variables of each rule implies a complete interval order type of the head variables, which we will call head IDB interval order type.

   (a) Let for each IDB P $\gamma_1^P, \ldots, \gamma_p^P$ be the head tree interval types of rules with head IDB P.
   (b) For each IDB P and each rule with head IDB P and tree interval type γ_i^P $(i = 1, \ldots, p)$, replace the head IDB by a new IDB P_i.
   (c) For each IDB P and each rule with an occurrence of P, determine for each occurrence the tree interval type of the variables in this occurrence, determine the corresponding tree interval type γ_i^P and then replace this occurrence of P by P_i. If no corresponding γ_i^P can be found, delete the rule.

The program created by this algorithm is clearly in the tree interval disjoint shape and equivalent to Π. Since the complete order type γ is in some cases converted to an incomplete order type by removing some atoms, different γ in the iteration over all order types may coincide, leading to identical rules. These rules may lead to a bigger program and hence a less efficient solving algorithm for DATALOG-TUPLE($\mathcal{A}$), but since they do not lead to inconsistencies, we may keep them.

The running time bounds and the bound on $|\Pi'|$ follows as in Lemma 6.37. □

After the transformation of the input program to the tree interval type disjoint form an initialization sequence can be created in polynomial time (in $|\Pi'|$) to solve DATALOG-NONEMPTINESS, analogously to Lemma 5.14 or Lemma 6.13 for colored orders. All IDB relations of the converted program are partitioned by the use of constants and order atoms, which is the form needed for generating the initialization sequence.

This is possible because the solving algorithm transfers all information important for DATALOG-NONEMPTINESS to the special shaped datalog program.

Lemma 7.17 *Let $\mathcal{A} = (A, <, c_1, \ldots, c_k)$ be a tree order with empty or infinite constant intervals. Let Π be an interval order type disjoint datalog program over $\mathcal{A}$.*

Then there exist an $i_s \leq n_I$ and a sequence $s = (\rho_0, \rho_1, \ldots, \rho_{i_s-1})$ of rules, such that after applying ρ_i at stage i for $i = 0, \ldots, i_s - 1$, the emptiness is determined, i.e. for all IDBs P of Π it holds that

$$P_{i_s}^{\Pi,\mathcal{A}} = \emptyset \quad \Rightarrow \quad P_{\infty}^{\Pi,\mathcal{A}} = \emptyset \ . \tag{7.1}$$

This sequence s can be computed in time $n_R \cdot n_I$.

Proof: (Sketch)

We create the sequence s by cycling through the rules n_I times, adding those rules to s which change an empty IDB to nonempty. Formally: $s = (\rho_0, \rho_1, \ldots, \rho_{i_s-1})$ such that there exist IDBs $P_1, \ldots, P_{i_s}$ with $(P_i)_i^{\Pi} = \emptyset$, and after applying ρ_i, $(P_i)_{i+1}^{\Pi} \neq \emptyset$ for $i = 0, \ldots, i_s - 1$.

We continue this process until no more rules can be applied to make an empty IDB nonempty, but this can happen at most n_I times, immediately leading to the time bound for the computation. Note that nonempty IDBs are never modified by s.

A short consideration shows why this is sufficient: Since the program is in tree interval disjoint form, for each IDB relation P the tuples contained in this relation have entries that are either equal to constants or are in an unbounded constant interval. By Lemma 7.13 all tuples with greater distance types (in the unbounded intervals) are automatically included. When using IDBs to instantiate a rule, then there are tuples available with arbitrarily large distance types if the IDB relation is nonempty.

Thus a rule of Π can be instantiated, if and only if all body IDBs are nonempty and the tree interval type of all head and body IDBs and the EDBs is consistent. A consistency check can be carried out in time polynomial in the rule length.

We now show property (7.1) by contradiction:

Let $U = \left\{ R \;\middle|\; R^{\Pi}_{i_s} = \emptyset \wedge R^{\Pi}_{\infty} \neq \emptyset \right\}$ be the set of IDBs changing to nonempty after s and assume $U \neq \emptyset$. Then for each $R \in U$ there exist an $i_R \in \mathbb{N}$ and a rule ρ_R with:

$$R^{\Pi}_{i_R} = \emptyset, \text{ and applying } \rho_R : \; R^{\Pi}_{i_R+1} \neq \emptyset \; .$$

Let $P \in U$ be the IDB with $i_P = \min \{ i_R \mid R \in U \}$. By the definition of U and by the choice of i_P, all $Q \in U \setminus \{P\}$ have to satisfy $Q^{\Pi}_{i_P} = \emptyset$. Since a rule can be applied if and only if all body IDBs are nonempty, the rule ρ_R cannot depend on them and can be applied in stage i_P leading to a sequence of rule applications making more IDBs nonempty, a contradiction to the construction of s. □

After creating the parts of the framework to solve DATALOG-NONEMPTINESS($\mathcal{A}$), we summarize them in a theorem:

Theorem 7.18 *On tree orders $\mathcal{A} = (A, <, c_1, \ldots, c_k)$ with finitely many constants, DATALOG-NONEMPTINESS can be solved in deterministic exponential time.*

Proof: Given the tree order $\mathcal{A}$, a tree order $\mathcal{A}'$ with infinite and empty constant intervals can be created in finitely many steps by Lemma 7.12. Based on $\mathcal{A}$ and these finitely many changes, Lemma 7.16 gives an algorithm for creating tree interval type disjoint versions of the input program in exponential time and of exponential size. Lemma 7.17 shows how to compute the nonemptiness of the relations of this program, directly leading to a solution for DATALOG-NONEMPTINESS($\mathcal{A}$). □

7.4 Tree Orders with Dense Parts

We will now have a look at some extensions of the tree order concept considered so far, beginning with tree orders that contain dense parts.

By a tree order with dense parts we will understand a tree order to which dense parts are added in a very controlled manner:

Definition 7.19 *Let $\mathcal{A} = (A, <)$ be a tree order and $x \in A$. Let $y \in A$ be a successor of x. A* tree order $\mathcal{A}' = (A', <')$ with a dense part D between x and y added *is the following structure:*

- $A' = A \cup D$
- $A \cap D = \emptyset$
- $<'$ *and* $<$ *agree on* A
- D *is a countable dense linearly ordered set for* $<'$
- *for all* $d \in D$, $x <' d <' y$.

A structure $\mathcal{A}' = (A', <')$ created from a tree order by adding (finitely many or countably infinitely many) dense parts between an element and one of its successors in $\mathcal{A}$, is called a tree order with dense parts.

A more general version of tree orders with dense parts could allow dense parts which are separated by branching points which are not part of the tree order, but of some superstructure. To avoid problems with completeness, we only allow dense parts as intervals between points of our original tree order, as in the definition above.

Lemma 7.20 *Let $\mathcal{A}' = (A', <)$ be a tree order with dense parts created from a tree order $\mathcal{A} = (A, <)$. Let $\mathcal{D} = (D, <)$ be a dense linear order (without maximum). Then DATALOG-NONEMPTINESS($\mathcal{A}$) and DATALOG-NONEMPTINESS($\mathcal{D}$) are equivalent.*

Proof: Since $D \subseteq A$, the nonemptiness of an IDB relation on D implies the nonemptiness on A immediately.

For the other direction we iteratively define a homomorphism $h : A \to D$.

We choose an arbitrary element $d_0 \in D$ and let $h(r) = d_0$ (where r is the minimal element of $\mathcal{A}$). We now choose an element $d_1 \in D$ with $d_0 < d_1$.

Let $S = S(r)$ be the set of successors of r in $\mathcal{A}$. For each $s \in S$, we let $h(s) = d_1$, and if there is a dense part P between r and s in $\mathcal{A}'$, we define h to be an arbitrary homomorphism from P into the set $D' = \{d \in D \mid d_0 < d < d_1\}$. Such a homomorphism exists, since D' as dense set is at least countably infinite and P is countable, and both are linear orders.

We now iterate this process in a breath-first-search manner, continuing with the successors of the elements in S and a $d_2 \in D$ with $d_1 < d_2$. Since D has countable infinite cardinality and is dense, we may continue this iterative definition for the whole tree order $\mathcal{A}'$. □

Corollary 7.21 *DATALOG-NONEMPTINESS($\mathcal{A}$) on tree orders with dense parts $\mathcal{A} = (A, <)$ is solvable in deterministic exponential time.*

Proof: This result uses the previous lemma and the result about DATALOG-NONEMPTINESS on (dense) linear orders, from Chapter 5: We just solve the problem on an arbitrary dense linear order. □

A combination of some tree order shapes considered so far are tree orders $\mathcal{A} = (A, <, c_1, \ldots, c_k)$ with dense parts. In such a tree order dense parts may occur either in parts of the structure that are not bounded from above by some constant or in a part bounded from above by some constant.

While in the first case we may simply replace the whole part by an infinite chain without affecting the nonemptiness of relations (similar to Lemma 7.20), dense parts bounded from above by a constant would force a change of the formalism introduced in Definition 7.11 and Lemma 7.12. While for a tree order without dense parts

only infinite unbounded or empty constant intervals occur we may now have infinite constant intervals bounded from above by a constant. Furthermore, in Lemma 7.12 the equivalence of the modified tree order was shown defining a homomorphism based on the distance to the root of the tree. With infinite bounded constant intervals the corresponding interval bounds or branching points would have to be used leading to a more technical approach. If this approach succeeds, the containment of DATALOG-NONEMPTINESS($\mathcal{A}$) in EXPTIME should follow directly.

7.5 Tree Order Variants

While the scope of this chapter was limited to commonly used discrete tree structures, tree-like orders also appear in less restricted form.

Infinite Branching

One variant is to drop the restriction of finite branching, i.e. any element may have infinitely many successors. To this case König's lemma does not apply any more and a third variant of tree order may appear. A tree order $\mathcal{A}$ may then either

- have an infinite chain C or
- have a bound $b \in \mathbb{N}$, such that all chains in $\mathcal{A}$ have length at most b (and there is a chain of length b) or
- may have chains of arbitrary finite length, i.e. for any $n \in \mathbb{N}$ there is a finite chain C in $\mathcal{A}$ of length at least n.

In the simpler case without constants one could show the equivalence of DATALOG-NONEMPTINESS($\mathcal{A}$) for tree orders $\mathcal{A}$ containing chains of arbitrary finite length and the ordered natural numbers $\mathcal{N} = (\mathbb{N}, <)$. It can easily be shown using a homomorphism that each relation nonempty on $\mathcal{A}$ can also be made nonempty on $\mathcal{N}$. But for the converse direction a straightforward homomorphism definition does not seem possible, since we would have to map the infinite chain in $\mathcal{N}$ to finite chains in $\mathcal{A}$.

But maybe a different approach can be used to show that DATALOG-NONEMPTINESS is equivalent on these two structures, using some finite initialization sequence. For this case some adaption of our arguments for linear orders would be necessary in which infinite branching does not lead to infinite type descriptions.

Colored Tree Orders

A similar problem arises in the case of tree orders with some additional monadic relations, i.e. colored tree orders. Using a color a set of chains of arbitrary finite length can be created from a tree order which has an infinite chain when no colors are present:

Example 7.22

We use an infinite chain C as the "backbone" of our order. At the i-th element of this chain we attach a finite chain F_i with i elements. We color all the finite chains F_i with the monadic relation, leaving C uncolored. See Figure 7.1 for a sketch of the structure.

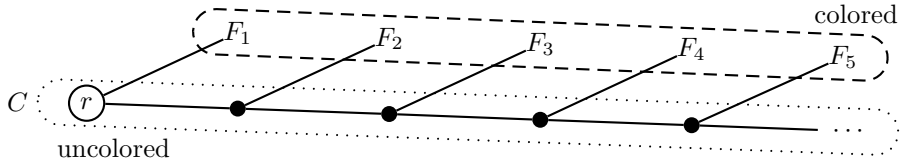


Figure 7.1: A colored tree order with backbone C

The colored tree order created this way will have an uncolored infinite chain, but it will have only finite chains of arbitrary length when considering the colored elements. Thus, the solution methods for both approaches (infinite chain, finite chains of arbitrary length) will have to be employed in some way.

Making things even more complicated, we could make the F_i infinitely long, but for each $i \in \mathbb{N}$, color only i elements of F_i, and use other colors to add more information.

At first glance, the methods introduced in this chapter combined with the methods from Chapter 6 about colored orders do not seem to be applicable in this context. Since it may be possible to distinguish different incomparable chains by colorings, the simple elimination of chains or branching points is unlikely to work. It could be possible to transfer the methods introduced for colored orders with more than one monadic relation to colored tree orders by using them on disjoint chains independently, but for this a reduction to finitely many chains would be needed which is even a problem in the small example demonstrated.

7.6 DATALOG-TUPLE

In this chapter we have only considered DATALOG-NONEMPTINESS. The main reason for that comes from the application of methods from Chapter 6 which gives a uniform bound only for DATALOG-NONEMPTINESS. In Example 6.44 we have demonstrated why the methods do not lead to a low uniform bound for DATALOG-TUPLE on linear orders.

The problem was there that if we convert the tuple entries of a DATALOG-TUPLE instance to constants, we may have to add a very big number of constants afterwards, depending on the value of the tuple entries. This makes the algorithm created from the structure dependent on the input instance, so it cannot be regarded as constant and the size of the type disjoint program may grow arbitrarily.

This argument can be transferred to the scenario of tree orders easily and even more complicated examples are imaginable which use branching points.

Similar to Example 7.22 constants and hence input tuple entries converted to constants may be used to mark parts of the structure with arbitrarily complex structure. Without these constants, these complex parts may be replaced by just an infinite chain. Input tuple entries could be used to "activate" them, increasing the complexity in an uncontrolled way.

As we discussed for colored orders in Section 6.5, we could transform the tuple of the DATALOG-TUPLE($\mathcal{A}$) instance into constants and solve DATALOG-NONEMPTINESS($\mathcal{A}'$) for the enhanced tree order $\mathcal{A}'$. Now a part of the structure $\mathcal{A}'$ is contained in the problem input, so instead of showing the existence of a transformed tree order with infinite or empty constant intervals and deriving a transformation algorithm for datalog programs into tree interval disjoint form as in Lemma 7.16, the structure conversion and the program transformation algorithm are dependent on the input and have to be considered as part of the problem solving algorithm.

Hence for solving DATALOG-TUPLE($\mathcal{A}$) the properties of $\mathcal{A}$ used in Lemmas 7.12 and 7.16 have to be decidable to derive the decidability of DATALOG-TUPLE($\mathcal{A}$) on a tree order $\mathcal{A}$.

7.7 Tree Orders and Negation

The idea we employed to simulate EDB negation in datalog programs over linear orders and colored linear orders was based on the replacement of $\neg(x < y)$ by rules with the atoms $y < x$ and $x = y$.

On tree orders it is unclear if such a simulation exists. The main problem is that we cannot translate $\neg(x < y)$ to a disjunction of order atoms or equalities, since also the case in which x and y are incomparable, would have to be considered and be defined by a *positive existential* formula.

7.8 Summary of this Chapter

Transforming some of the methods from Chapters 5 and 6 we derived some upper bounds for tree orders which are based on infinite trees.

We have shown that DATALOG-NONEMPTINESS($\mathcal{A}$) is in EXPTIME for the following cases:

- tree orders $\mathcal{A} = (A, <)$
- tree orders with constants $\mathcal{A} = (A, <, c_1, \ldots, c_k)$
- tree orders with dense parts $\mathcal{A} = (A, <)$

Open Problems

Tree orders are a first step from linear orders to partial orders. For our results of tree orders, we used the presence of chains in tree orders, which occur in a very controlled way. In fact, the set $x^<$ of elements smaller than a given element x is always a finite chain in tree orders. For general partial orders, we loose this property and for any element x we may not only have incomparable elements that are comparable with an element from $x^<$, but also from $x^>$.

A transfer of the results to partial orders seems to involve some technical difficulties, but is seems possible. So an interesting open question remaining is:

Question 7.1 *Can the result of the EXPTIME-completeness of DATALOG-NONEMPTINESS be transferred to partial orders?*

One first step in this direction might be the application of our methods to tree orders $\mathcal{A} = (A, <, c_1, \ldots, c_k)$ with dense parts.

Question 7.2 *Is DATALOG-NONEMPTINESS($\mathcal{A}$) in EXPTIME for tree orders $\mathcal{A} = (A, <, c_1, \ldots, c_k)$ with dense parts?*

Most of our methods were based on homomorphisms, so using results about homomorphisms on (partial) orders could be useful here.

Chapter 8

Constraint Satisfaction

Outline of this Chapter

In this chapter we first transfer some of our results for datalog on infinite structures to constraint satisfaction problems on infinite linear orders. After that we modify some of our ideas from the discussion of the undecidability of datalog on successor structures in Chapter 4 to show that there is no dichotomy of constraint satisfaction problems on infinite structures between NP (or EXPTIME) and the undecidable cases.

8.1 CSPs

Constraint satisfaction problems occur in many areas of computer science, like database theory, graph theory, bio-informatics, computational linguistics and artificial intelligence. Widely used they have different definitions of which we will choose an abstract algebraic definition.

Let Γ (the *template*) be a structure over some vocabulary σ. Then CSP(Γ) is the following decision problem:

Definition 8.1 ***CSP***(Γ)*:*

Instance: *A finite structure S of the same relational vocabulary σ as Γ*

Question: *Is there a homomorphism $h : S \to \Gamma$?*

A different, but equivalent characterization is: Given a set of relations $R_1, \ldots, R_k$, the problem is to decide whether a formula $\varphi(x_1, \ldots, x_k)$ which is a conjunction of atoms with symbols from $\{R_1, \ldots, R_k\}$ has a satisfying variable assignment.

For a boolean universe these relations $R_1, \ldots, R_k$ correspond to boolean clauses and the whole problem to the well known NP-complete satisfiability problem (SAT). From this context CSPs are sometimes called "generalized satisfiability problems".

Besides the *non-uniform* version CSP(Γ), there is also a *uniform* version CSP(Δ, Γ) which asks whether there is a homomorphism $h : \Delta \to \Gamma$, but the non-uniform CSP has been focus of research in the last years, for finite templates ([Sch78; HN90; TM93; FV99; KV98; Jea98; KV00]), as well as for infinite templates ([BN03; Bod04; BD06; BN06; BK06]). Some recent results [Gro03; Gro07] examine CSP(Δ, _), where the template is the domain of the homomorphism.

Example 8.2 *Many graph theoretical problems can easily be formulated as constraint satisfaction problems. For example the graph-two-colorability problem:*

GRAPH-2-COLORABILITY

Instance: *Undirected Graph $G = (V, E)$*

Question: *Can we color the vertices of G with two colors such that no two adjacent vertices have the same color?*

This problem can be naturally modelled as CSP, see Figure 8.1.

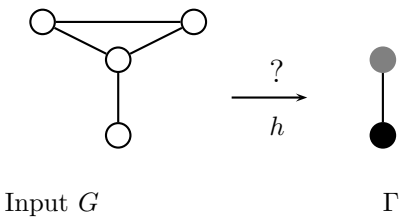


Figure 8.1: Graph-Two-Colorability as CSP

Similarly, graph-k-colorability can be modelled as CSP(K_k), where K_k is the complete graph with k vertices.

Example 8.3 *As an example of a CSP with an infinite template, have a look at CSP($(\mathbb{Z}, <)$), the linear orders over the integer numbers.*

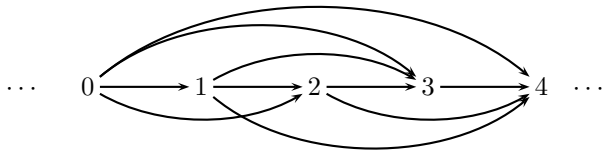


Figure 8.2: $(\mathbb{Z}, <)$ represented as a digraph

The representation of $(\mathbb{Z}, <)$ as a digraph shows an obvious connection to the topological sorting of a graph (see Figure 8.2). A topological sorting is possible, if

and only if a digraph is acyclic, making the equivalence of CSP$(\mathbb{Z}, <)$ to the following problem immediate:

DIGRAPH-ACYCLICITY

Instance: *Directed Graph $G = (V, E)$*

Question: *Is G acyclic, i.e. does not contain a directed cycle?*

In fact, we use this problem and its rather low complexity to solve CSPs on linear orders.

Complexity of CSPs

For finite templates CSP is clearly in the complexity class NP, since we can guess a homomorphism and verify it in polynomial time. However, for some templates Γ, CSP(Γ) is in polynomial time (tractable) and for some Γ, CSP(Γ) is NP-complete. For some restricted domains, i.e. templates with two elements [Sch78], and template with three elements [Bul03], it has been shown that there is a dichotomy and a CSP is either tractable or NP-complete. On templates that are finite graphs, there is also a dichotomy between tractable and NP-complete (see [HN90]). This leads to the dichotomy conjecture for finite templates in [FV99]:

Conjecture: For finite Γ, CSP(Γ) is either tractable or NP-complete.

The results mentioned use an algebraic approach; see, e.g. [PJ97; Jea98].

On infinite structures, CSP(Γ) may have a much higher complexity than NP, for example the relational structure $\Gamma = (\mathbb{N}, +, *, 0, 1)$ has an undecidable CSP(Γ), which follows from the undecidability of Hilbert's 10th problem on Diophantine Equations (see [Bod04]).

On the other hand, there are classes of structures whose CSP can be shown to be in NP, like the finitely constrained structures Γ which are characterized by a finite set of forbidden substructures (of a first order expansion of Γ), as shown in [Bod04].

Recent publications transfer the algebraic approach with closure conditions to infinite domains to show the tractability in these cases (see [BN03; BD06]) and a dichotomy has been shown for equality constraint languages in [BK06].

There are several different links between constraint satisfaction problems and datalog. Trying to solve constraint satisfaction problems using datalog, one observes that some problems are solvable with datalog programs having some bounded IDB arity k and some bounded number ℓ of variables per rule. This leads to the notion of bounded width (k, ℓ). In [FV99] it was shown how to create a canonical program for finite CSPs with bounded width (k, ℓ) to solve the CSP, which was transferred to infinite ω-categorical templates in [Bod04]. In [KV98] the connections between constraint satisfaction problems and datalog are examined via pebble games, for datalog programs of a fixed maximal number of rule variables and on finite structures.

Constraint satisfaction problems can be seen as single rule non-recursive datalog programs, where the body variables of the rule are exactly the head variables. We use this approach to show some bounds for constraint satisfaction problems on infinite orders, but of course the results are quite rough, since they do not use any special properties of non-recursive datalog.

8.2 Transfer of Order Results to CSP

Considering that CSPs can be seen as single rule datalog programs consisting of a non-recursive rule we now try to transfer some datalog results to the context of constraint satisfaction problems. The simplest case we research, is the case CSP$((A, <))$, constraint satisfaction on (infinite) linear orders.

Thus letting $\Gamma = (A, <)$ be a strict infinite linear order[1], CSP(Γ) is the question: Given a finite structure S over vocabulary $\sigma = \{<^2\}$, is there a homomorphism $h : S \to (A, <)$?

An equivalent more datalog like form of this problem is: Given a formula φ which is a conjunction of $<$-atoms, is there a satisfying assignment to the variables $\bar{x}$ of $\varphi(\bar{x})$ with elements of A? This, however is DATALOG-NONEMPTINESS($\mathcal{A}$) for the following single rule datalog program Π_φ:

$$P(\bar{x}) \leftarrow \varphi(\bar{x}).$$

We may now easily apply the results from Chapter 5 to get a singly exponential time bound for solving this problem. But our discussion in that chapter includes ideas for solving DATALOG-NONEMPTINESS for non-recursive rules, which lead to a much tighter upper bound. In the proofs of Lemma 5.5 and Lemma 5.27, we used a graph structure to represent all order atoms contained in a rule and to examine the satisfiability by checking this graph for cycles. We may define the graph representing the formula φ as:

Definition 8.4 ***(Order Graph of a Formula φ)***

Let $\varphi(\bar{x})$ be a conjunction of order atoms $x_i < x_j$, where $\bar{x} = (x_1, \ldots, x_k)$ are the variables of $\varphi(\bar{x})$. Then the order graph $G_\varphi = (V, E)$ of φ is the directed graph defined as:

$$V := \{x_1, \ldots, x_k\}$$
$$E := \{(x_i, x_j) \mid x_i < x_j \text{ occurs in } \varphi(\bar{x})\}$$

Then on strict infinite linear orders the equivalence of the following two properties is immediate:

1. $\varphi(\bar{x})$ is satisfiable over $(A, <)$

[1] For non-strict orders, the problem is trivial.

2. G_φ contains no cycle.

If G_φ contains a cycle $x_1, x_2, \ldots, x_1$, the variable assignment will have to fulfill $x_1 < x_2 < \cdots < x_1$, which is impossible. If on the other hand G_φ is acyclic, a topological sorting of G_φ is possible, defining an order on the vertices such that for each two vertices u and v with $u < v$ there is no edge from v to u. We can find a satisfying variable assignment by assigning ascending values corresponding to the topological order.

Using the graph representation, we have reduced CSP$(A, <)$ to the problem of digraph-acyclicity, which is in the complexity class NL, the class of nondeterministic logarithmic space [2].

Lemma 8.5 *CSP($\mathcal{A}$) for an infinite linear order $\mathcal{A} = (A, <)$ can be solved in nondeterministic logarithmic space, and hence is included in PTIME.*

The more sophisticated cases of orders we have considered so far, were orders aggregated with constants and monadic relations. In the context of constraint satisfaction problems, the scope is limited to purely relational structures, where constants do not occur. We may, however, integrate the constants into our structure as monadic relations which is the case we will take a brief look at.

Let $\mathcal{A} = (A, <, M_1, \ldots, M_m)$ be a colored order (without constants), as defined in Section 6.1. Then we can solve each CSP($\mathcal{A}$) instance as if it was a single rule datalog program Π over $\mathcal{A} = (A, <, M_1, \ldots, M_m)$. The methods introduced in Chapter 6 will first transform the structure $\mathcal{A}$ to an interval sequence order $\mathcal{A}' = (A', <', I_1, \ldots, I_n, c_1, \ldots, c_\ell)$, transform Π into an interval order type disjoint version Π' which may be exponentially larger than Π, and then solve DATALOG-NONEMPTINESS($\mathcal{A}$) for this program in time exponential in the length of Π, leading to:

Lemma 8.6 *For a colored infinite linear order $\mathcal{A} = (A, <, M_1, \ldots, M_m)$, CSP($\mathcal{A}$) is included in EXPTIME.*

This transformation makes it necessary to check the nonemptiness of several IDBs of Π', testing whether at least one of them is nonempty, although the original CSP and hence the program Π only dealt with one relation. If the time for solving a CSP on colored orders could be decreased, then it would be necessary to reduce the number of IDBs in advance, but without the full construction of the exponentially larger program or even checking all possible order types. It seems very unlikely, that our approach will lead to a lower upper bound.

[2] [Jon75] shows CYCLE $\in$ NL, hence digraph-acyclicity $\in$ co-NL, and from e.g. [Pap94] NL=co-NL, hence digraph-acyclicity $\in$ NL.

8.3 EXPTIME-Undecidable Non-Dichotomy

The complexity results in this chapter all have lead to upper bounds of at most singly exponential complexity, in some cases even less. In the last case considered, it seems likely that using a different approach a much lower upper bound could be shown. Other results for constraint satisfaction with infinite template research some cases in which an NP upper bound can be shown and a dichotomy between tractable (PTIME) and NP-complete is of interest and shown for some cases (see [Bod04], [BD06], [BK06]) or conjectured, following the dichotomy conjecture for CSP with finite templates from [FV99].

Contrasting these computational easy cases, the following example was shown to be undecidable ([Bod04]):

Example 8.7 *Consider CSP*$((\mathbb{N}, +, *, 0, 1))$ *which implements arithmetics over natural numbers as relational structure. In this case '+' and '*' are ternary relations containing all tuples satisfying* $x + y = z$ *and* $x * y = z$*, respectively;* 0 *and* 1 *are singleton relations containing only* 0 *and* 1*.*

The equivalence of CSP$((\mathbb{N}, +, *, 0, 1))$ *to Hilbert's 10th problem on Diophantine Equations shows the undecidability.*

This observation may lead to the idea of some other dichotomy here, in this case a dichotomy for CSP on infinite structures between NP or EXPTIME complexity and undecidable cases. We will adapt some ideas from the similar context concerning a possible dichotomy of datalog on infinite structures (see Chapter 4) which will lead to a similar result: We will create a family of structures, each with a CSP in some iterated exponential complexity classes. Unfortunately the bounds shown are not tight, but have an exponential gap between upper and lower bound, but also then a dichotomy may not exist.

For creating CSPs with various time complexity bounds we will reuse some ideas from Section 4.3 and create a structure to a given function $f : \mathbb{N} \to \mathbb{N}$. In Section 4.3 we created some successor structures of different length of which depending on the program length n and hence on the number of variables in a rule, only those of maximal length $f(n)$ could be used. On these structure the Turing machine simulation from Chapter 3 was used to show that any computation which finishes in time $f(n)$ could be simulated. But the Turing machine simulation heavily depends on the recursive structure of datalog and basically the recursion depth is the number of steps simulated. Being confronted with a formalism like CSP without recursion, we have to change to a different approach. The basic idea is that we will encode all encodings of starting configurations of the universal Turing machine that lead to a computation accepting in at most $f(n)$ steps into our structure $\mathcal{C}_f$. Solving a problem decidable in deterministic time $f(n)$ then means to check if the starting configuration of a computation can be mapped to $\mathcal{C}_f$ which only contains the starting configurations of accepting computations.

Since we are dealing with purely relational structures, we have to find a way to encode bit strings into relations. A common approach in mathematical logic is

to take a successor structure as index set, often enriched with an order, and define monadic relations on this set — for an alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$, one for each σ_i which contains all positions with occurrences of σ_i. These representation of strings as relational structures are called word models (see [EF99]) and are usually defined with an order on the index set instead of a successor relation. Since word models are usually considered on more powerful concepts than positive existential logic, a successor can easily be defined from an order relation, using quantification and negation. On the other hand, for defining an order relation using a successor relation, some recursive concept like fixed point operators is needed which is why a given order relation is more convenient than a successor in many cases.

Example 8.8 *Let $w = 101011$ be a string over the alphabet $\Sigma = \{0, 1\}$. As a relational structure, this string can be represented by $\mathcal{S} = (\{1, \ldots, 6\}, \mathrm{suc}^2, C_0^1, C_1^1)$. The relation* suc *is interpreted as the binary successor relation on the index set $S := \{1, \ldots, 6\}$, while $C_0 = \{2, 4\}$ and $C_1 = \{1, 3, 5, 6\}$ contain the positions of the occurrences of* 0 *and* 1*.*

We could also extend $\mathcal{S}$ with some relations F and L, denoting the first and last position which are otherwise inaccessible without negation or universal quantification: $\mathcal{S}' = (S, \mathrm{suc}^2, C_0^1, C_1^1, F^1, L^1)$ with $F = \{1\}$ and $L = \{6\}$.

We will build our structure $\mathcal{C}_f$ from disjoint substructures of this kind, each encoding a Turing machine starting configuration.

Definition 8.9 *Let $f(n) : \mathbb{N} \to \mathbb{N}$ be a computable function. Let $w_1, w_2, w_3, \ldots \in \{0, 1\}^*$ be the binary encodings of the starting configurations of the universal Turing machine that accept in at most $f(n)$ steps.*

We define the structure $\mathcal{C}_f = (C, S^2, F^1, L^1, C_0^1, C_1^1)$ to be the structure with an infinite universe C which is a disjoint union of the following substructures:

- *For each w_i, $\mathcal{C}_f$ contains a finite substructure of some elements $s_1^i, \ldots, s_{|w_i|}^i$ defining w_i:*

$$\begin{aligned} s_j^i &\in C_0 && \text{for all } j \text{ with } w_i[j] = 0 \\ s_j^i &\in C_1 && \text{for all } j \text{ with } w_i[j] = 1 \\ s_1^i &\in F \\ s_{|w_i|}^i &\in L \\ (s_j^i, s_{j+1}^i) &\in S && \text{for } j = 1, \ldots, |w_i| - 1 \end{aligned}$$

- *A successor loop with one element $s \in F$:*

$$(s, s) \in S, \quad s \in C_0, \quad s \in C_1, \quad s \in F$$

- *A successor loop with one element $s' \in L$:*

$$(s', s') \in S, \quad s' \in C_0, \quad s' \in C_1, \quad s' \in L$$

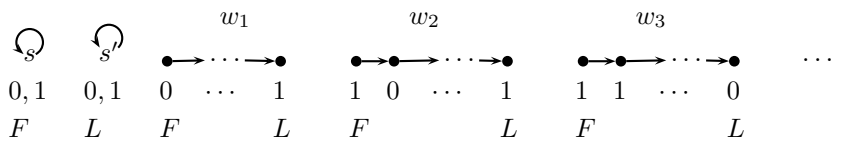


Figure 8.3: The Structure $\mathcal{C}_f$ for some f

In Figure 8.3 the structure is shown as a directed graph, where S is the edge relation.

All relations in this structure are decidable, and hence the whole structure is decidable. A decision algorithm, if there exists a successor substructure of length n encoding the starting configuration of some computation, which runs at most $f(n)$ steps and ends in the accepting state, is simply a simulation of the universal Turing machine for $f(n)$ steps. A closer look to CSP($\mathcal{C}_f$) will show more details how to decide some properties of $\mathcal{C}_f$:

Lemma 8.10 *For each computable function $f : \mathbb{N} \to \mathbb{N}$ with $2^n \in O(f(n))$, CSP($\mathcal{C}_f$) is included in DTIME(f^2).*

Proof: The input is an instance $\mathcal{T} = (T, S^2, F^1, L^1, C_0^1, C_1^1)$, a finite structure with the same vocabulary as $\mathcal{C}_f$. We now have to determine a homomorphism $h : \mathcal{T} \to \mathcal{C}_f$. We describe an algorithm to determine if there is such a homomorphism in time $O(2^n \cdot f(n))$.

We interprete $\mathcal{T}$ as directed graph $G = (T, S)$ with edge relation S and each vertex labeled according to the containment in the sets F, L, C_0 and C_1. Let G_u be an undirected version of G, i.e. for $G = (V, E)$, we let $G_u = (V, E \cup E^T)$ and copy all the labels F, L, C_0 and C_1 to G_u. Without loss of generality we assume that G_u has only one connection component, since we can carry out the following steps for each connection component separately.

If T does not contain an element $x \in T$ which is included in F, we may let $h(x) := s'$ for all $x \in T$. This mapping is a homomorphism, since $(s', s') \in S$, $s' \in L$, $s' \in C_0$ and $s' \in C_1$. So each membership constraint valid in T will be preserved by h. An analogous construction is possible, if no element $x \in T$ is included in L mapping all elements to s.

In the other case there are elements from both F and L included in T. If there is a directed path from an element in L to an element in F, then clearly there is no homomorphism $h : \mathcal{T} \to \mathcal{C}_f$, since $\mathcal{C}_f$ does not contain such a path, and the answer would be negative.

If, on the other hand, there is no (un-) directed path from an element in F to an element in L in G_u, we may divide the Graph into components: G_F containing all

vertices reachable from F in G_u and G_L containing all other vertices. Defining

$$h(x) = \begin{cases} s, & \text{if } x \in G_F \\ s', & \text{otherwise,} \end{cases}$$

this mapping h is the homomorphism we asked for. In the substructure of $\mathcal{C}_f$ with universe $\{s\}$, all membership constraints are satisfiable, with the exception of $s \in L$. In the substructure on $\{s'\}$ all membership constraints are satisfiable not containing F.

Note, that all these cases considered so far may be checked in linear time.

Since in the graph G elements from F and L are related by successor-expressions (relation S), they have to be mapped to one of the starting configuration encodings in $\mathcal{C}_f$. A quick check now assures that there are no elements from F with ingoing edges or elements from L with outgoing edges and that there are no vertices v with $v \in C_0 \cap C_1$, since this does not occur in any of the starting configuration encodings and directly leads to a negative answer for the instance $\mathcal{T}$.

It now remains to check if G can be homomorphically mapped to a consistent encoding of some binary string and if this string encodes one of the accepting starting configurations.

Let $p = (x_1, \ldots, x_m)$ be the shortest path in G_u from F to L, i.e. the shortest path with $x_1 \in F$ and $x_m \in L$. This path can be found in linear time, e.g. using breadth first search, and it will be the base for the encoding in G.

If p only uses edges from E, i.e. forward edges, this path can be used as the index set for the encoding of a string by C_0 and C_1 and we only have to check that other vertices in G are labeled consistent. If p uses backward edges from E^T, the consistency check must be used to determine if p is the index set of a string encoding and of which string:

G has to be mapped to some of the string encodings in $\mathcal{C}_f$ by the homomorphism h. Since each of these encodings is some kind of linked list structure, we now check if G can be mapped to a list structure.

We first calculate the distance of each vertex to x_1, the first vertex of p, using a breadth first search on G_u, starting with $d(x_1) = 0$. When visiting a vertex u and considering all neighbors of u during the search, we extend d to each neighbor v by:

$$\begin{aligned} d(v) &:= d(u) + 1 \quad \text{if } (v, u) \in E \\ d(v) &:= d(u) - 1 \quad \text{if } (u, v) \in E \end{aligned}$$

If a conflict occurs when setting a new $d(v)$ value, both (v, u) and (u, v) are in E, which is a cycle and renders a homomorphic map to $\mathcal{C}_f$ impossible and we may stop the process with a negative answer. If for an already visited vertex v, the value $d(v)$ contradicts the new value calculated from neighbor u, we have two paths of different length between x_1 and u, also making a homomorphism impossible and we stop the algorithm with a negative answer. Similar for negative $d(x_i)$ values, which correspond to vertices with a path to an element in F.

With these distances calculated we may check if all vertices having the same distance to x_1 may be mapped to the same element to get a homomorphism. This can be checked by considering the set D_d of all vertices v having the same distance value $d(v) = d$. If D contains vertices in C_0 and C_1, then they cannot be mapped to one vertex of $\mathcal{C}_f$, since the only vertices in $\mathcal{C}_f$ satisfying this property are s and s' which we can exclude, as they are not connected to both F and L. After assuring that all vertices $v \in F$ in G have distance $d(v) = 0$ and all vertices $v \in L$ in G have the same distance d_L, we know that G may be homomorphically mapped to the representation of a binary string. Otherwise we stop the algorithm with negative answer.

Once we know, that we may map G to a relational representation of a binary string, we derive this string w from G: For each distance $i = 0, \ldots, d_L$, we consider all vertices v with $d(v) = i$. If there is a vertex $v \in C_0$, we let $w_i = 0$, if there is a vertex $v \in C_1$, we let $w_i = 1$. Otherwise, we let $w_i =?$.

To check if G maps homomorphically to a binary encoded starting position in $\mathcal{C}_f$, we insert all different combinations of 0 and 1 for the ?-positions in the string w, leading to a set $\{s_1, \ldots, s_m\}$, which may be exponentially large in d_L, since for each ?-position there are two possible values. See Figure 8.4 for an example.

For each of these strings $s_1, \ldots, s_m$, we check if it is a valid encoding of a starting configuration of the universal Turing machine and then simulate the machine run for at most $f(d_L)$ steps.

If one of the computations is accepting, we have found a starting configuration of a run accepting in at most $f(d_L)$ steps and hence there must be an encoding of this starting configuration in $\mathcal{C}_f$. But then we can map G homomorphically to this encoding and the answer of the algorithm is positive.

Most of the algorithm described in this proof has linear running time. The only part, that is not linear, is the last part: There, we have at most 2^{d_L} computations, which run in time $O(f(d_L))$ each[3], leading to a total running time of $O(2^{d_L} \cdot f(d_L)) \subseteq O(f(n)^2)$. □

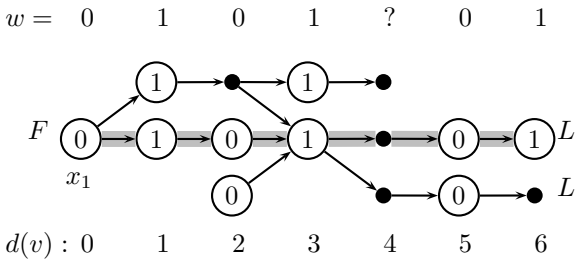
While the upper bound needed some considerations, the lower bound is quite immediate:

Lemma 8.11 *For each function $f : \mathbb{N} \to \mathbb{N}$, the problem CSP($\mathcal{C}_f$) is hard for DTIME($f$).*

Proof: Let P be a problem from DTIME(f). Then there exists a Turing machine M such that for each P-instance x, M accepts in deterministic time $f(|x|)$, if and only $x \in P$. As reduction from P to CSP($\mathcal{C}_f$), we generate the starting configuration s for the simulation of M with input x as a relational encoded binary string. Then there exists a homomorphism $h : s \to \mathcal{C}_f$ if and only if $x \in P$. This is the reduction showing the result. □

If we combine the two Lemmas 8.10 and 8.11, we get the following corollary:

[3]The check if the starting configuration is valid can be done in time polynomial in d_L.



This graph G may be mapped to relational encodings of the string $w = 0101?01$. The vertices belonging to C_0 are labeled with 0, those belonging to C_1 with 1, while the filled vertices do not belong to any of these relations. The membership in F and L is denoted by the corresponding letters F and L. The shaded edges are the edges of path p.

Figure 8.4: Example for a graph G corresponding to valid string encodings

Corollary 8.12 *As in Section 4.3, let*

$$\mathrm{tow}(r,i) := 2^{\cdot^{\cdot^{2^i}}}\Big\}r$$

and

$$r\mathrm{EXP} := \mathrm{DTIME}\left(2^{\cdot^{\cdot^{2^{n^k}}}}\Big\}r\right) .$$

Then CSP$(\mathcal{C}_{\mathrm{tow}(r,\cdot)})$ *is hard for* $(r-1)$EXP *and CSP*$(\mathcal{C}_{\mathrm{tow}(r,\cdot)})$ *is contained in* rEXP *, or shorter:*

$$(r-1)\mathrm{EXP} \subseteq \mathit{CSP}(\mathcal{C}_{\mathrm{tow}(r,\cdot)}) \subseteq r\mathrm{EXP}$$

The strict inclusion of the iterated exponential time complexity classes rEXP shows that a dichotomy between EXPTIME complexity and undecidable cases of CSP on infinite structures cannot exist.

8.4 Summary of this Chapter

We have transferred some of our datalog results by treating CSPs as single-rule nonrecursive datalog programs. While some results seem rather tight, like the NL complexity of CSP$(A, <)$, the EXPTIME upper bound derived from colored orders seems quite high.

We have then transferred some results of our discussion about the EXPTIME-undecidability-dichotomy-conjecture to the context of CSPs and have derived a similar result, this time only exhibiting a singly exponential gap between lower and upper complexity bound.

Open problems

Question 8.1 *What is the complexity of CSP*$(A, <, M_1, \ldots, M_m)$*?*

Question 8.2 *Can an improved upper complexity bound be shown by refining our method?*

Of course, as in Chapter 4:

Question 8.3 *Can the exponential gap in Section 8.3 be closed?*

Chapter 9

Conclusion and Open Problems

In this chapter we will give a overview of the results of this thesis. A more detailed review of the results of each chapter can be found at the end of the corresponding chapter, but here we will put the results in a more general context. Each chapter also includes some open problems which the author of this works considers most interesting among them.

9.1 Conclusion

9.1.1 Complexity and Undecidability

The general scope of this thesis are the decidability and complexity of datalog, to be more precise, of DATALOG-NONEMPTINESS and DATALOG-TUPLE. We group the most important of these results by complexity and problem type.

1. Undecidability:

DATALOG-NONEMPTINESS and DATALOG-TUPLE on successor structures are undecidable: Chapter 4

2. Lower Bounds:

DATALOG-NONEMPTINESS and DATALOG-TUPLE on strict linear orders have EXPTIME-hard complexity: Chapter 3

3. Upper Bounds (DATALOG-NONEMPTINESS):

On linear orders DATALOG-NONEMPTINESS has EXPTIME complexity, even with finitely many constants (Chapter 5), and even on colored orders with finitely many monadic relations: Chapter 6

As an application of the methods, in Chapter 7, these results are transferred to tree orders, leading to an EXPTIME upper bound for tree orders (with and without constants, discrete and also with additional dense parts).

4. **Upper Bounds (DATALOG-TUPLE):**

   With some natural assumptions about the representation of the infinite structures, DATALOG-TUPLE($\mathcal{A}$) is EXPTIME-complete on dense linear orders, even with constants. DATALOG-TUPLE is decidable on linear orders: Chapter 5

5. **Boundedness:**

   Datalog programs are uniformly bounded on linear orders: Chapter 5

6. **Hierarchy:**

   In Chapter 4 we have shown that there is a set of infinite structures such that for each level $\mathcal{C}$ of the exponential time hierarchy there is a structure on which DATALOG-NONEMPTINESS is included in $\mathcal{C}$ and hard for the class one level lower, leaving no room for a EXPTIME-undecidable-dichotomy.

Our investigation of the complexity of DATALOG-NONEMPTINESS over strict linear orders including colored orders, together with the undecidability results for successor structures gives a complete overview of the complexity depending on the arity of additional EDB relations allowed:

Complexity of DATALOG-NONEMPTINESS on Linear Orders		
Additional EDBs:	Complexity	Result
none (arity 0)	EXPTIME-complete	Chapter 5
monadic (arity 1)	EXPTIME-complete	Chapter 6
maximal arity ≥ 2	undecidable	Chapter 4

9.1.2 Concepts Introduced

The concepts developed in these investigations can also be applied in a different context, e.g. when computing datalog queries:

1. **Distance Types:**

   Types modeling the order and distance of entries of tuples in IDB relations have been shown to be sufficient to describe datalog computations on linear orders, even with constants: Chapter 5

2. **Interval Orders and Interval Types:**

   Types partitioning the structure into finitely many disjoint intervals and distance types have been shown to be a sufficient replacement of order, constants and monadic relations of colored orders to decide DATALOG-NONEMPTINESS: Chapter 6

   An application of this concept for tree orders is demonstrated in Chapter 7.

3. Type Disjoint Programs:

Type disjoint programs are transformation of datalog programs which contain all the distance type or interval order and distance type information needed to solve DATALOG-NONEMPTINESS and DATALOG-TUPLE. Different versions were defined in Chapters 5, 6 and 7.

While the distance type concept may lead to practical applications and is suitable for both problems, the interval order and distance type concept only gives a solution method for DATALOG-NONEMPTINESS. Why DATALOG-TUPLE is not considered in this context, is discussed in Chapter 6. The concept of type disjoint program may be interesting in other settings, where describing types can be defined.

9.1.3 Constraint Satisfaction Problems

In Chapter 8 we transferred the results on datalog to the context of constraint satisfaction problems:

1. Linear Orders:

CSP$((A, <))$ for a linear order $(A, <)$ is in NL$\subseteq$PTIME.

2. Colored Orders:

CSP$((A, <, M_1, \ldots, M_m))$ for a colored infinite linear order $(A, <, M_1, \ldots, M_m)$ is in EXPTIME.

3. Hierarchy:

We have shown that there is a set of infinite structures such that for each level $\mathcal{C}$ of the exponential time hierarchy there is a structure on which CSP is included in $\mathcal{C}$ and hard for the class one level lower, leaving no room for a EXPTIME-undecidable-dichotomy for CSP on infinite structures.

Since our methods highly concentrate on the recursive structure of datalog, they seem to be of minor relevance for CSPs, as the first two results show. The third result shows an approach which may be a promising tool in different contexts for refuting such dichotomies.

9.2 Open Problems

The examination of the complexity of DATALOG-TUPLE on orders has not lead to tight bounds. Of course it would be interesting to narrow or even close this gap:

1. Can we show a higher lower bound for DATALOG-TUPLE on orders? (Chapter 3)

2. Can the computable upper bound for DATALOG-TUPLE on orders be lowered, maybe even to singly exponential? (Chapter 5)

During the investigation of the undecidability of datalog, successor structures and some variants have been considered, leading to the obvious question:

> What are the properties of a structure needed, such that it can be used to define an infinite successor structure using datalog? (Chapter 4)

During the discussion of datalog on orders and colored orders, similar questions arise:

1. On which structures can a type concept similar to the *distance types* be established to solve both DATALOG-NONEMPTINESS and DATALOG-TUPLE?
2. On which structures can a type concept similar to the *interval order types* be established to solve DATALOG-NONEMPTINESS?
3. On which structures does DATALOG-NONEMPTINESS have EXPTIME complexity, or is at least decidable?

Bibliography

[Abi89] Abiteboul, S.: Boundedness is undecidable for datalog programs with a single recursive rule. In: *Information Processing Letters*, volume 32:pp. 281–287, 1989.

[AHV95] Abiteboul, S.; Hull, R.; Vianu, V.: *Foundations of Databases*. Addison-Wesley, 1995.

[AK00] Ash, C.J.; Knight, J.F.: *Computable structures and the hyperarithmetical hierarchy*. Elsevier, Amsterdam, 2000. ISBN 0-444-50072-3.

[All82] Allen, J. F.: Maintaining knowledge about temporal intervals. In: *Comm. ACM*, volume 26:pp. 832–843, 1982.

[BD06] Bodirsky, M.; Dalmau, V.: Datalog and constraint satisfaction with infinite templates. In: Durand, B.; Thomas, W., editors, *Proceedings of the 23rd Annual Symposium on Theoretical Aspects of Computer Science*, volume 3884 of *Lecture Notes in Computer Science*, pp. 646–659. Springer-Verlag, 2006.

[BG00] Blumensath, A.; Grädel, E.: Automatic structures. In: *lics*, volume 00:pp. 51–62, 2000. ISSN 1043-6871. doi:http://doi.ieeecomputersociety.org/10.1109/LICS.2000.855755.

[BG04] Blumensath, A.; Grädel, E.: Finite presentations of infinite structures: Automata and interpretations. In: *Theory of Computing Systems*, volume 37(6):pp. 641–674, December 2004.

[BK06] Bodirsky, M.; Kara, J.: The complexity of equality constraint languages. In: *Proceedings of the International Computer Science Symposium in Russia (CSR06)*, volume 3967 of *LNCS*, pp. 114–126. 2006.

[BN03] Bodirsky, M.; Nešetřil, J.: Constraint satisfaction with countable homogeneous templates. In: *LNCS*, volume 2803:p. 4457, 2003. URL `citeseer.ist.psu.edu/article/bodirsky03constraint.html`.

[BN06] Bodirsky, M.; Nešetřil, J.: Constraint satisfaction with countable homogeneous templates. In: *Journal of Logic and Computation*, volume 16:pp. 359–373, 2006.

[Bod04] Bodirsky, M.: *Constraint satisfaction with infinite domains*. Ph.D. thesis, Humboldt-Universit"at zu Berlin, 2004.

[Bul03] Bulatov, A.: A dichotomy theorem for constraints on a three-element set. In: *In Proceedings of 33rd IEEE International Symposium on Multiple-Valued Logic (ISMVL'02)*, pp. 343–351. Tokyo, Japan, 2003.

[Bur98] Burris, S. N.: *Logic for Mathematics And Computer Science*. Prentice Hall, 1998. ISBN 0-13-285974-2.

[Büc60] Büchi, J. R.: On a decision method in restricted second order arithmetic. In: *Proc. Internat. Congr. on Logic, Methodology and Philosophy of Science*, pp. 1–11. Stanford University Press, 1960.

[Cau96] Caucal, D.: On infinite transition graphs having a decidable monadic theory. In: auf der Heide, F. Meyer; Monien, B., editors, *Proceedings of the 23th International Colloquium on Automata, Languages and Programming (ICALP'96)*, volume 1099, pp. 194–205. Springer, Berlin-Heidelberg-New York, 1996. URL `citeseer.ist.psu.edu/caucal96infinite.html`.

[CGKV88] Cosmadakis, S. S.; Gaifman, H.; Kanellakis, P. C.; Vardi, M. Y.: Decidable optimization problems for database logic programs (preliminary report). In: *Proceedings of the 20th ACM Symposium on Theory of Computing*, pp. 477–490. 1988.

[CGZ07] Cohen, S.; Gil, J.; Zarivach, E.: Datalog programs over infinite databases, revisited, 2007. To appear.

[Chi90] Chisholm, J.: Effective model theory vs. recursive model theory. In: *The Journal of Symbolic Logic*, volume 55(3):pp. 1168–1191, September 1990.

[Cou89] Courcelle, B.: The monadic second-order logic of graphs ii: Infinite graphs of bounded width. In: *Math. Systems Theory*, volume 21:pp. 187–221, 1989.

[DEGV97] Dantsin, E.; Eiter, T.; Gottlob, G.; Voronkov, A.: Complexity and expressive power of logic programming. In: *IEEE Conference on Computational Complexity*, pp. 82–101. 1997. URL `citeseer.ist.psu.edu/article/dantsin97complexity.html`.

[Dic13] Dickson, L.E.: Finiteness of the odd perfect and primitive abundant numbers with discitnct factors. In: *American Journal of Mathematics*, volume 3:pp. 413–422, 1913.

[Die00] Diestel, R.: *Graphentheorie*. Springer, Berlin, 2nd edition, 2000. ISBN 3-540-67656-2.

[EF99] Ebbinghaus, H.-D.; Flum, J.: *Finite Model Theory*. Springer, berlin, 2nd edition, 1999. ISBN 3-540-65758-4.

[EFT98] Ebbinghaus, H.-D.; Flum, J.; Thomas, W.: *Einführung in die mathematische Logik*. Spektrum, Heidelberg, 4th edition, 1998. ISBN 3-8274-0130-5.

[EGNR98] Ershov, Y. L.; Goncharov, S. S.; Nerode, A.; Remmel, J.B.: *Handbook of Rectusive Mathematics*. North-Holland, 1998.

[FV99] Feder, T.; Vardi, M.: The computational structure of monotone monadic snp and constraint satisfaction: A study through datalog and group theory. In: *SIAM Journal on Computing*, volume 28(1):pp. 57–104, 1999.

[GG98] Grädel, E.; Gurevich, Y.: Metafinite model theory. In: *Information and Computation*, volume 140:pp. 26–81, 1998.

[GJ79] Garey, M.; Johnson, D.: *Computers and Intractability*. Freeman, 1979.

[GMSV93] Gaifman, H.; Mairson, H.; Sagiv, Y.; Vardi, M.Y.: Undecidable optimization problems for database logic programs. In: *Journal of the ACM*, volume 40(3):pp. 683–713, 1993.

[Gro03] Grohe, M.: The complexity of homomorphism and constraint satisfaction problems seen from the other side. In: *FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, p. 552. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-2040-5.

[Gro07] Grohe, M.: The complexity of homomorphism and constraint satisfaction problems seen from the other side. In: *J. ACM*, volume 54(1):pp. 1–24, 2007. ISSN 0004-5411. doi:http://doi.acm.org/10.1145/1206035.1206036.

[GS] Grohe, M.; Schwandtner, G.: The complexity of datalog on linear orders. Submitted to LMCS.

[HKMV95] Hillebrand, Gerd G.; Kanellakis, Paris C.; Mairson, Harry G.; Vardi, Moshe Y.: Undecidable boundedness problems for datalog programs. In: *Journal of Logic Programming*, volume 25(2):pp. 163–190, 1995. URL `citeseer.ist.psu.edu/article/hillebrand95undecidable.html`.

[HMS06] Harwood, W.; Moller, F.; Setzer, A.: Weak bisimulation approximants. In: *Proceedings of CSL'06*, LNCS, pp. 365–379. Szeged, Hungary, 2006.

[HN90] Hell, P.; Nešetřil, J.: On the complexit of h-coloring. In: *Journal of Combinatorial Therory, Series B*, volume 48:pp. 92–110, 1990.

[Hod97] Hodges, W.: *A shorter model theory.* Cambridge University Press, New York, NY, USA, 1997. ISBN 0-521-58713-1.

[Jea98] Jeavons, P.: On the algebraic structure of combinatorial problems. In: *Theoretical Computer Science*, volume 200(1–2):pp. 185–204, 1998. URL `citeseer.ist.psu.edu/jeavons98algebraic.html`.

[Jon75] Jones, N. D.: Space-bounded reducibility among combinatorial problems. In: *Journal of Computer and System Sciences*, volume 11:pp. 68–85, 1975.

[KG94] Kanellakis, Paris C.; Goldin, Dina Q.: Constraint programming and database query languages. In: *Theoretical Aspects of Computer Software*, pp. 96–120. 1994. URL `citeseer.ist.psu.edu/kanellakis94constraint.html`.

[KJJ04] Krokhin, A.; Jeavons, P.; Jonsson, P.: Constraint satisfaction problems on intervals and lengths. In: *Siam J. Discrete Math.*, volume 17(3):pp. 453–477, 2004.

[KLP00] Kuper, G.; Libkin, L.; Paredaens, J., editors: *Constraint Databases.* Springer, 2000.

[Kre99] Kreutzer, S.: *Descriptive Complexity Theory for Constraint Databases.* Master's thesis, Rheinisch-Westfälische Technische Hochschule Aachen, 1999.

[KV95] Kolaitis, Ph.G.; Vardi, M.Y.: On the expressive power of datalog: tools and a case study. In: *Journal of Computer and System Sciences*, volume 51(1):pp. 110–134, 1995.

[KV98] Kolaitis, Ph. G.; Vardi, Moshe Y.: Conjunctive-query containment and constraint satisfaction. In: *Journal of Computer and System Sciences*, pp. 205–213. 1998. URL `citeseer.ist.psu.edu/kolaitis98conjunctivequery.html`.

[KV00] Kolaitis, Ph. G.; Vardi, Moshe Y.: A game-theoretic approach to constraint satisfaction. In: *AAAI/IAAI*, pp. 175–181. 2000. URL `citeseer.ist.psu.edu/kolaitis00gametheoretic.html`.

[Mar99] Marcinkowski, J.: Achilles, turtle, and undecidable boundedness problems for small datalog problems. In: *Siam J. Comput.*, volume 29(1):pp. 231–257, 1999.

[Mei96] Meiri, I.: Combining qualitative and quantitative constraints in temporal reasoning. In: *Artificial Intelligence*, volume 87:pp. 343–385, 1996.

[Mos74] Moschovakis, Y. N.: *Elementary Induction on Abstract Structures*. North Holland, 1974.

[MS83] Muller, D.; Schupp, P.: Groups, the theory of ends, and context-free languages. In: *Journal of Computer and System Sciences*, volume 26:pp. 295–310, 1983.

[MS85] Muller, D.; Schupp, P.: The theory of ends, pushdown automata, and second order logic. In: *Theoretical Computer Science*, volume 37:pp. 51–75, 1985.

[NB95] Nebel, B.; Bürckert, H.-J.: Reasoning about temporal relations: a maximal tractable subclass of Allen's interval algebra. In: *Journal of the ACM*, volume 42, 1995.

[Pap94] Papadimitriou, C.: *Computational Complexity*. Addison-Wesley, New York, 1994. ISBN 0-201-53082-1.

[PJ97] Pearson, J. K.; Jeavons, P. G.: A survey of tractable constraint satisfaction problems. Technical Report CSD-TR-97-15, Royal Holloway University of London, 1997. URL `citeseer.ist.psu.edu/pearson97survey.html`.

[PSV96] Papadimitriou, C. H.; Suciu, D.; Vianu, V.: Topological queries in spatial databases. In: *Journal of Computer and System Sciences*, pp. 81–92. 1996. URL `citeseer.ist.psu.edu/papadimitriou96topological.html`.

[Rev93] Revesz, P.Z.: A closed form evaluation for datalog queries with integer (gap)-order constraints. In: *Theoretical Computer Science*, volume 116:pp. 117–149, 1993.

[Rev95] Revesz, P.Z.: Constraint databases: A survey. In: *Semantics in Databases*, pp. 209–246. 1995. URL `citeseer.ist.psu.edu/revesz98constraint.html`.

[Sch78] Schaefer, T. J.: The complexity of satisfiablity problems. In: *Proceedings of the 10th ACM Symposium on Theory of Computing*, pp. 216–226. 1978.

[Sch01] Schöning, U.: *Algorithmik*. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2001.

[SS82] Sebelik, J.; Stepanek, P.: Horn clause programs for recursive functions. In: Clark, K.; Tarnlund, S.-A., editors, *Logic Programming*. Academic Press, 1982.

[SV89] Sagiv, Y.; Vardi, M. Y.: Safety of datalog queries over infinite databases. In: *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'89)*, pp. 160–171. ACM Press, Philadelphia, Pennsylvania, United States, 1989.

[TM93] T.A.Feder; M.Y.Vardi: Monotone monadic snp and constraint satisfaction. In: *Proceedings fo the 25th ACM Symposium on Theory of Computing*, pp. 612–622. 1993.

[Tom95] Toman, D.: *Foundations of Temporal Query Languages*. Ph.D. thesis, Kansas State University, 1995.

[Ull88a] Ullman, J. D.: *Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[Ull88b] Ullman, J. D.: *Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1988.

[Var82] Vardi, M. Y.: Complexity of relational query languages. In: *Proc. of 14th Symposium on Theory of Computing*, volume 53, pp. 137–146. 1982.

List of Figures

List of Tables

Selbständigkeitserklärung

Hiermit erkläre ich, dass

- ich die vorliegende Dissertationsschrift selbständig und ohne unerlaubte Hilfe verfasst habe,
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze,
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin (veröffentlicht im Amtlichen Mitteilungsblatt Nr. 34/2006) bekannt ist.

Goetz Schwandtner